

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Sistemas de Gestión de Workflows para Plataformas Serverless

Trabajo Fin de Máster

Máster Universitario en Computación en la Nube y de Altas Prestaciones / Cloud and High-Performance Computing

AUTOR/A: Smiller Sánchez, José Eduardo

Tutor/a: Moltó Martínez, Germán

CURSO ACADÉMICO: 2024/2025



UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Departamento de Sistemas Informáticos y Computación

Universitat Politècnica de València

Sistemas de Gestión de Workflows para Plataformas Serverless

Trabajo Fin de Máster

Máster Universitario en Computación en la Nube y de Altas Prestaciones

Autor: José Eduardo Smiller Sánchez

Tutores: Germán Moltó Martínez, Amanda Calatrava Arroyo

2024-2025

Resumen

Este Trabajo de Fin de Máster explora el uso de herramientas de orquestación de *workflows* en entornos *serverless*, con especial atención a su integración con OSCAR, una plataforma *open-source* basada en Kubernetes. Se realiza un análisis comparativo de soluciones como Synapse, SonataFlow, AWS Step Functions, Airflow, Temporal, Cadence y Prefect, evaluando criterios como compatibilidad con contenedores, enfoque declarativo, integración con Kubernetes y facilidad de uso. Tras descartar opciones por diversas limitaciones, se selecciona Argo Workflows como solución idónea. Se desarrolla una implementación práctica que demuestra cómo Argo Workflows puede gestionar servicios en OSCAR, incluyendo flujos de procesamiento de imágenes y análisis de sentimiento. El trabajo valida la viabilidad técnica de la integración, destacando la modularidad, escalabilidad y reproducibilidad del sistema en entornos *serverless* distribuidos.

Palabras clave: Serverless, Workflows, Argo, OSCAR

Abstract

This Master's Thesis explores the use of workflow orchestration tools in serverless environments, with a special focus on their integration with OSCAR, an open-source platform based on Kubernetes. A comparative analysis of solutions such as Synapse, SonataFlow, AWS Step Functions, Airflow, Temporal, Cadence, and Prefect is performed, evaluating criteria such as container compatibility, declarative approach, Kubernetes integration, and ease of use. After ruling out options due to various limitations, Argo Workflows is selected as the ideal solution. A practical implementation is developed that demonstrates how Argo Workflows can manage services in OSCAR, including image processing workflows and sentiment analysis. The work validates the technical feasibility of the integration, highlighting the system's modularity, scalability, and reproducibility in distributed serverless environments.

Keywords: Serverless, Workflows, Argo, OSCAR

Tabla de contenidos

1. Introducción 9
2. Estado del arte 12
   2.1 Sistemas de Gestión de *Workflows* en Entornos *Serverless* 12
   2.2 OSCAR 13
      2.2.1 Componentes del entorno 13
   2.3 Aspectos considerados en la evaluación 15
      2.3.1 Enfoque declarativo 15
      2.3.2 Compatibilidad con Kubernetes 15
      2.3.3 Integración con Contenedores 15
   2.4 Evaluación de Synapse y SonataFlow 16
      2.4.1 Synapse 16
      2.4.2 SonataFlow 17
   2.5 Evaluación de otras herramientas 18
      2.5.1 AWS Step Functions 18
      2.5.2 Apache Airflow 19
      2.5.3 Temporal 20
      2.5.4 Cadence 21
      2.5.5 Prefect 23
   2.6 Argo Workflows 23
      2.6.1 Arquitectura general 24
      2.6.2 Ventajas de utilizar Argo Workflows en OSCAR 25
   2.7 Conclusión 26
3. Desarrollo 27
   3.1 Entorno de trabajo 27
      3.1.1 Descripción del entorno utilizado 27
      3.1.2 Otras arquitecturas posibles 28
   3.2 Instalación de Argo Workflows en OSCAR 29
   3.4 Integración 35
      3.3.1 Ejecución de servicios mediante la API de OSCAR 36
      3.3.2 Ejecución de servicios mediante la carga de un archivo 38

3.3.3 Patrones de ejecución asíncrona 41
4. Validación 45
4.1 Objetivo 45
4.2 Entorno 45
4.3 *Workflow* 46
4.3.1 Pasos del *workflow* 47
4.3.2 Diseño y definición del *workflow* 50
4.4 Ejecución 56
4.4.1 Ejecución correcta del flujo 56
4.4.2 Desempeño 58
5. Conclusiones 61
5.1 Trabajos futuros 62
6. Referencias 63

Índice de figuras

Figura 1: Arquitectura y componentes de OSCAR 14
Figura 2: Arquitectura básica de Apache Airflow, fuente [16] 20
Figura 3: Máquina de estado en Temporal, fuente [17] 21
Figura 4: Arquitectura de un servicio Cadence, fuente [19] 22
Figura 5: Componentes principales de Argo Workflows 24
Figura 6: Interacción de la ejecución de un workflow en OSCAR 30
Figura 7: Manifiesto del Ingress que permite visualizar la interfaz gráfica de Argo Workflows 32
Figura 8: Comando utilizado para crear la cuenta de servicio argo-user 32
Figura 9: Manifiesto del rol que permitirá a argo-user administrar workflows 33
Figura 10: Crear binding del rol y la cuenta de servicio 33
Figura 11: Secreto que albergará el token de la cuenta 33
Figura 12: Comando para obtener el token del secreto previamente creado 33
Figura 13: Pantalla de inicio de la interfaz gráfica de Argo Workflows 34
Figura 14: Solicitud HTTP a la API de Argo Workflows 34
Figura 15: Comando utilizado para crear el secreto para la ejecución del servicio Cowsay 36
Figura 16: Definición de un workflow que obtiene el token a partir de un secreto para llamar un servicio 37
Figura 17: Workflow que convierte un grupo de imágenes a escala de grises 39
Figura 18: Resultado de la ejecución del workflow image-processing-b7qjr 40
Figura 19: Los archivos resultantes de la ejecución del workflow image-processing-wg9v2 40
Figura 20: Workflow que convierte imágenes a escala de grises para posteriormente clasificar esas imágenes 41
Figura 21: Ejecución del workflow classify-plants 41
Figura 22: Solicitud HTTPS que resume un nodo del workflow 43
Figura 23: Solicitud HTTPS que detiene un nodo del workflow 43
Figura 24: Comparación entre los distintos métodos de espera o respuesta ante la finalización del servicio 44
Figura 25: Definición simple del workflow de validación 45
Figura 26: Recursos del clúster OSCAR 46
Figura 27: Entrada y sálida del workflow 47
Figura 28: Definición detallada del workflow de validación 48
Figura 29: Patrón fan-out/fan-in 49
Figura 30: Representación gráfica del DAG definido en el workflow 51
Figura 31: Definición FDL del servicio recolector de datos de Reddit 52
Figura 32: La tarea que orquesta el análisis de sentimientos 54
Figura 33: Paso de visualización de datos 56

Figura 34: Ejecución del flujo de análisis de sentimientos sentiment-analysis-workflow-2q5lt 57
Figura 35: Archivos generados para el paso data-split-task del workflow sentiment-analysis-workflow-2q5lt 57
Figura 36: El archivo resultante del workflow sentiment-analysis-workflow-2q5lt se actualiza en el repositorio 58
Figura 37: Dashboard interactivo de visualización de datos 58
Figura 38: Tiempos de ejecución del workflow 59

1. Introducción

El auge de la computación en la nube ha transformado la forma en que las organizaciones desarrollan y despliegan aplicaciones, impulsando modelos de ejecución más flexibles y escalables. En este contexto, la computación *serverless* ha cobrado relevancia como una alternativa eficiente que permite ejecutar funciones en respuesta a eventos sin necesidad de gestionar servidores o infraestructura subyacente. La computación *serverless* es un modelo de programación y arquitectura en el que fragmentos de código se ejecutan en la nube sin ningún control sobre los recursos donde se ejecuta el código. Esto no significa que no haya servidores, sino simplemente que el desarrollador debe delegar la mayoría de las preocupaciones operativas, como la provisión de recursos, el monitoreo, el mantenimiento, la escalabilidad y la tolerancia a fallos, al proveedor de la nube [1]. Este modelo de ejecución bajo demanda ofrece ventajas como la reducción de costes operativos y una escalabilidad automática, lo que ha fomentado su adopción en múltiples ámbitos, desde aplicaciones web hasta procesamiento de datos en tiempo real.

Sin embargo, la orquestación de tareas en entornos *serverless* representa un desafío significativo. Por diseño, las funciones sin servidor no tienen estado y no pueden comunicarse directamente con otras funciones concurrentes. Esta característica no es adecuada para *workflows*, que requieren inherentemente que las tareas se comuniquen entre sí [2]. Dado que las aplicaciones pueden requerir la ejecución coordinada de múltiples funciones con dependencias de datos entre ellas, se destaca la necesidad de herramientas que permitan definir y gestionar *workflows* eficientes.

En entornos tradicionales, los sistemas de gestión de *workflows* han sido ampliamente utilizados para estructurar y automatizar procesos, pero su adaptación a plataformas *serverless* aún es un área en desarrollo [2]. Este trabajo busca abordar esta problemática explorando el uso de Argo Workflows [3] como herramienta de orquestación para gestionar *workflows* en entornos sin servidor.

Argo Workflows es una plataforma *open-source* diseñada para la ejecución de *workflows* en Kubernetes [4]. Su arquitectura basada en contenedores permite definir *workflows* mediante archivos YAML, facilitando la orquestación de tareas complejas en entornos distribuidos. Gracias a su capacidad para integrarse con múltiples servicios en la nube, Argo Workflows se presenta como una solución adecuada para gestionar la ejecución de funciones *serverless* en plataformas como AWS Lambda [5] y entornos *on-premises* como OSCAR, una plataforma de código abierto para respaldar el modelo de computación *serverless* impulsado por eventos para aplicaciones de procesamiento de datos [6].

Este Trabajo de Fin de Máster (TFM) tiene como objetivo general explorar el uso de Argo Workflows como herramienta de orquestación para gestionar la ejecución de funciones *serverless*, con el fin de garantizar una coordinación eficiente de tareas en entornos distribuidos.

Para alcanzar este objetivo, se plantean los siguientes objetivos específicos:

- Analizar la interoperabilidad de Argo Workflows con diferentes servicios y herramientas utilizadas en arquitecturas *serverless*.

- Evaluar el rendimiento de Argo Workflows en la ejecución de *workflows* complejos bajo distintas condiciones de carga.

- Comparar Argo Workflows con otras herramientas de orquestación de workflows, considerando aspectos como la facilidad de uso y adecuación al paradigma *serverless*.

- Demostrar mediante un caso de uso práctico la viabilidad de implementar Argo Workflows con plataformas de ejecución de funciones como OSCAR para la automatización de *pipelines*.

- Investigar patrones de ejecución asíncrona y concurrencia en la ejecución de funciones orquestadas desde Argo Workflows.

Con ello, este trabajo busca contribuir a una mejor comprensión del papel que juegan los sistemas de orquestación en arquitecturas *serverless*, y ofrecer una evaluación detallada sobre la aplicabilidad de Argo Workflows en este contexto.

Tras la introducción, el resto del documento está estructurado en cinco capítulos principales. En el capítulo 2, se presenta el estado del arte, donde se analizan diversos sistemas de gestión de *workflows* en entornos *serverless*. Así mismo, se detallan los aspectos clave considerados para la comparación y se expone con detalle el sistema seleccionado para este trabajo: Argo Workflows.

El capítulo 3 está dedicado al desarrollo e implementación del sistema. Se describe el entorno de trabajo utilizado, incluyendo tanto la arquitectura como los componentes principales, y se explica el proceso de instalación de Argo Workflows sobre OSCAR. Además, se aborda la integración entre ambos sistemas, mostrando diferentes mecanismos de ejecución de servicios y patrones de orquestación.

En el capítulo 4 se lleva a cabo la validación del sistema propuesto. Se detallan los objetivos de la validación, el entorno empleado para las pruebas y el *workflow* utilizado. También se presentan los resultados de ejecución, con un análisis del comportamiento observado y del desempeño del sistema bajo distintas condiciones.

El capítulo 5 presenta las conclusiones del trabajo, destacando los logros alcanzados, las limitaciones encontradas y posibles líneas de trabajo futuro.

Finalmente, el capítulo 6 incluye las referencias bibliográficas que respaldan el contenido desarrollado a lo largo del documento.

2. Estado del arte

2.1 Sistemas de Gestión de *Workflows* en Entornos *Serverless*

La gestión de *workflows* es un aspecto fundamental en el desarrollo de arquitecturas *serverless*, ya que permite la coordinación de múltiples funciones y servicios en entornos altamente dinámicos. Un sistema de gestión de *workflows* o *Workflow Management System* (*WFMS*) es una herramienta que facilita la definición, ejecución y monitorización de *workflows*, asegurando que las tareas se ejecuten en el orden correcto y con las dependencias adecuadas, definidas en la lógica del *workflow* [22].

En entornos *serverless*, la gestión de *workflows* presenta desafíos debido a la naturaleza temporal y sin estado de las funciones. A diferencia de arquitecturas tradicionales, donde los procesos pueden mantener estado de forma persistente, en *serverless* cada función se ejecuta de manera aislada, sin memoria de ejecuciones anteriores [2]. Esto obliga a que la orquestación del *workflow* se maneje externamente, mediante herramientas que permitan encadenar funciones, manejar errores, controlar condiciones de ejecución y gestionar dependencias temporales o de datos entre tareas. Además, dado que la ejecución se activa típicamente por eventos, los *workflows* deben ser lo suficientemente flexibles como para adaptarse a flujos no lineales, a ejecuciones concurrentes y ejecuciones condicionales.

El uso de plataformas *serverless* para la ejecución de *workflows* científicos introduce una serie de desafíos de arquitectura que deben ser considerados. Entre ellos se destacan la necesidad de encapsular el código y las dependencias de cada función, la gestión de tareas que requieren acceso a sistemas de archivos, y las limitaciones del entorno de ejecución, como el tiempo máximo de ejecución o el espacio de almacenamiento temporal disponible. Además, la naturaleza efímera y escalable de las funciones sin servidor dificulta el manejo del estado, la coordinación entre tareas concurrentes y el control de errores, lo cual obliga a recurrir a soluciones como colas de mensajes, almacenamiento intermedio o mecanismos de registros externo para mantener la trazabilidad del *workflow* [8]. Estos aspectos evidencian que, si bien el modelo *serverless* ofrece beneficios como elasticidad y menor coste operativo, también exige rediseñar tanto el motor de orquestación como las tareas individuales para que se ajusten a este nuevo paradigma.

En este contexto, han surgido diversas soluciones para la orquestación de *workflows* en plataformas *serverless*, entre ellas destacan Synapse, Apache SonataFlow y Argo Workflows, entre otras. En la propuesta inicial de este trabajo se consideró investigar Synapse y Apache SonataFlow como posibles alternativas para la gestión de *workflows* en combinación con OSCAR. Sin embargo, a

continuación se exponen las razones por las cuales ambas herramientas fueron descartadas como opciones viables para el desarrollo de este proyecto.

2.2 OSCAR

OSCAR es una plataforma de código abierto que soporta el modelo de computación *serverless* basado en eventos para aplicaciones de procesamiento de datos. Se puede implementar automáticamente en múltiples proveedores y gestores en la nube, e incluso en dispositivos de bajo consumo, para crear aplicaciones sin servidor de procesamiento de datos basadas en eventos altamente paralelas a lo largo del continuo computacional. Estas aplicaciones se ejecutan en entornos de ejecución personalizados proporcionados por contenedores Docker que se ejecutan en clústeres elásticos de Kubernetes. También se integra con el marco SCAR [10], que soporta un modelo de programación de computación de alto rendimiento (del inglés High Throughput Computing, HTP) para crear aplicaciones sin servidor de procesamiento de datos basadas en eventos altamente paralelas que se ejecutan en entornos de ejecución personalizados proporcionados por contenedores Docker que se ejecutan en AWS Lambda y AWS Batch [28].

2.2.1 Componentes del entorno

Un despliegue de OSCAR está conformado por varios componentes dentro de un clúster Kubernetes que permiten la ejecución de los servicios como contenedores de manera escalable y asíncrona. A continuación, se detallan algunos de ellos.

MinIO

MinIO es un servidor de almacenamiento de objetos de alto rendimiento, distribuido y compatible con la API de Amazon S3 [24]. Al cargarse archivos en un *bucket* específico, se generan eventos que luego OSCAR captura y convierte en *jobs* de Kubernetes.

Para interactuar con MinIO, gestionar archivos y consultar objetos, entre otras opciones, se puede utilizar MinIO Client (mc), una herramienta de línea de comandos diseñada para interactuar con servidores de MinIO o compatibles con el API de S3 [25]. En este proyecto, se accede a dicha herramienta a través de los contenedores Docker basados en la imagen *bitnami/minio-client*, disponible en Docker Hub, que incluye el cliente previamente instalado y listo para su uso en entornos automatizados.

OSCAR Manager

OSCAR Manager es la API principal del sistema y se encarga de la gestión e integración de los distintos componentes del clúster [26]. A través de esta API es posible crear y ejecutar servicios de forma tanto síncrona como asíncrona, así como acceder a registros de ejecución, estado del clúster y otra información relevante. Esta funcionalidad resulta clave para la integración de los servicios dentro de los *workflows* orquestados mediante Argo Workflows.

OSCAR Dashboard

El Dashboard de OSCAR es una interfaz gráfica web diseñada para usuarios finales [27]. Esta herramienta facilita la observación de registros de ejecución, la depuración de errores y el monitoreo general del sistema de forma visual e intuitiva. Durante el desarrollo del proyecto, el *dashboard* ha sido útil para validar rápidamente el estado de los servicios y analizar los resultados de las ejecuciones.

CLUES

CLUES (CLUster Energy Saving for HPC and cloud) es un sistema de gestión de energía para clústeres de computación de alto rendimiento e infraestructuras en la nube. Su función principal es apagar los nodos internos del clúster cuando no se utilizan y, a la inversa, encenderlos cuando son necesarios. El sistema CLUES se integra con el *middleware* de gestión del clúster, como un sistema de colas por lotes o un sistema de gestión de infraestructura en la nube, mediante diferentes conectores [31].

Knative

Knative es una plataforma de código abierto construida sobre Kubernetes que facilita la ejecución de aplicaciones *serverless*, basadas en eventos y empaquetadas como contenedores. Su principal objetivo es simplificar el despliegue y la gestión de servicios que escalan automáticamente en función de la demanda. Knative permite que los contenedores se ejecuten como servicios *serverless*, escalando a cero cuando no reciben tráfico y aumentando automáticamente los recursos asignados cuando se incrementa la carga, lo que lo convierte en una solución eficiente y adaptable para entornos dinámicos [32].

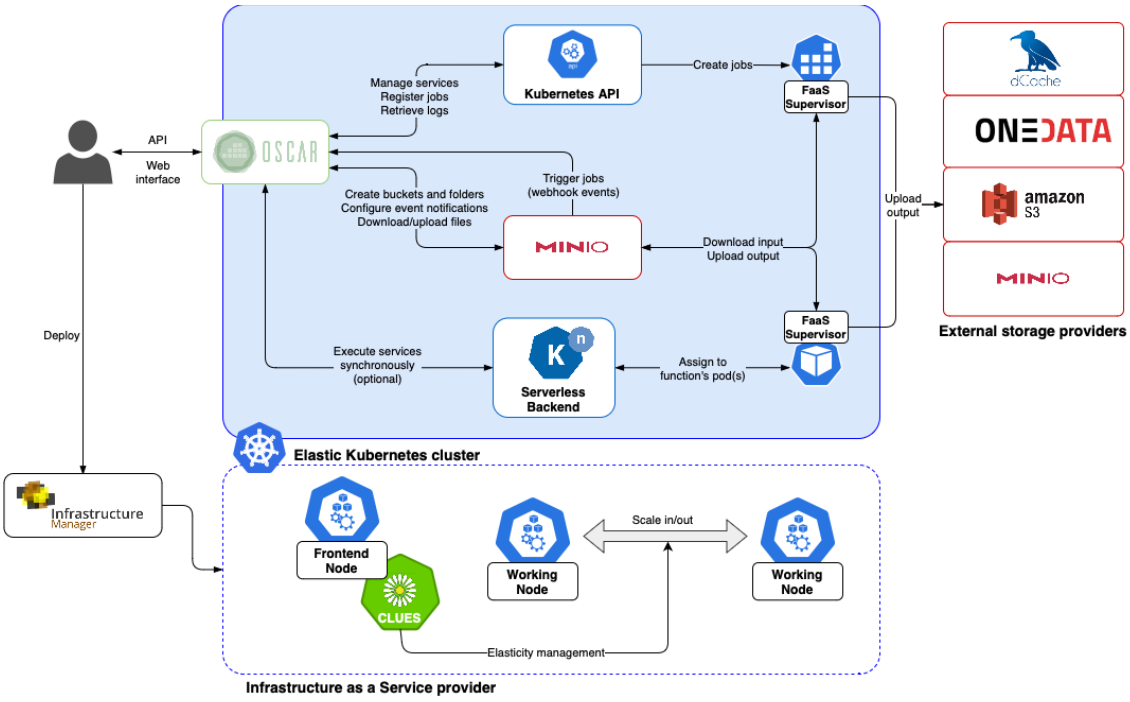


Figura 1: Arquitectura y componentes de OSCAR

En la Figura 1 se muestra con mayor detalle la arquitectura y componentes de un despliegue OSCAR, incluyendo los servicios externos a los que se puede conectar el clúster y demás aspectos técnicos y de infraestructura relevantes.

2.3 Aspectos considerados en la evaluación

Antes de seleccionar la herramienta de orquestación más adecuada para este trabajo, fue necesario definir los aspectos técnicos que guiaran en la evaluación de distintas plataformas. Estos aspectos se definieron teniendo en cuenta tanto los objetivos del proyecto como las características particulares de la arquitectura de OSCAR. La selección debía contemplar no solo la compatibilidad técnica con este entorno, sino también factores como la facilidad de uso, la escalabilidad, la reproducibilidad y la curva de aprendizaje. A continuación, se detallan los principales criterios que guiaron la evaluación y que justifican la elección final de Argo Workflows como la herramienta central para la gestión de *workflows* en este trabajo.

2.3.1 Enfoque declarativo

El enfoque declarativo es relevante para este proyecto porque permite describir el comportamiento general del *workflow* (los pasos, sus relaciones y condiciones de ejecución) mediante archivos YAML, sin necesidad de implementar la lógica de orquestación como código imperativo. Aunque algunos pasos del *workflow* pueden requerir la ejecución de *scripts* o código específico (por ejemplo, para mover archivos entre *buckets* o realizar llamadas HTTP a servicios), estos fragmentos de lógica se encapsulan dentro de tareas individuales, mientras que la administración del flujo completo (qué tareas se ejecutan, en qué orden y bajo qué condiciones) se mantiene separada y se define de forma declarativa. Esta separación de responsabilidades permite que los *workflows* sean fácilmente editables y compartibles, sin requerir que el usuario tenga conocimientos avanzados de orquestación, y se alinea con prácticas modernas como GitOps, mejorando la portabilidad y mantenibilidad del sistema.

2.3.2 Compatibilidad con Kubernetes

Dado que OSCAR está construido sobre Kubernetes y utiliza contenedores como unidad básica de ejecución, era muy importante que la plataforma seleccionada ofreciera compatibilidad con esta tecnología. Esto incluye el despliegue como recurso de Kubernetes, el uso de Custom Resource Definitions (CRDs) para representar *workflows*, y la capacidad de integrarse de forma natural con la gestión de *pods*, volúmenes, secretos y demás componentes del clúster. Se evaluó la capacidad de las herramientas de integrarse en el clúster sin necesidad de requerir adaptaciones complejas o despliegues adicionales que complicasen su uso.

2.3.3 Integración con Contenedores

Un aspecto importante en la ejecución de *workflows* modernos es la capacidad de ejecutar cada paso como un contenedor independiente. Esta integración

permite encapsular la lógica de cada tarea dentro de un entorno controlado, facilitando la modularidad, el aislamiento y la reproducibilidad. Además, trabajar con contenedores permite que los *workflows* puedan desplegarse en distintos entornos sin cambios significativos en la configuración, lo cual es especialmente importante cuando se busca portabilidad y consistencia en la ejecución [9].

2.4 Evaluación de Synapse y SonataFlow

Inicialmente, se consideró el uso de Synapse y SonataFlow por su implementación de la Serverless Workflow Specification (SWS), un estándar promovido por la Cloud Native Computing Foundation (CNCF) que busca proporcionar una representación unificada y portable para la definición de *workflows* en entornos sin servidor [11]. El principal objetivo de esta especificación es permitir que los *workflows* puedan definirse de forma declarativa, mediante archivos YAML o JSON, y ser ejecutados en diferentes plataformas compatibles sin necesidad de reescribirlos o ajustarlos a implementaciones particulares. Adoptar herramientas que siguen este tipo de estándares facilita la portabilidad de los *workflows* entre entornos y su integración con distintas herramientas de desarrollo que implementen el estándar, además de fomentar prácticas abiertas y sostenibles a largo plazo.

Esta orientación basada en estándares era especialmente relevante para este proyecto, ya que uno de los objetivos es promover *workflows* reutilizables y fácilmente reproducibles en distintas infraestructuras y proyectos. No obstante, durante el proceso de evaluación se encontraron limitaciones prácticas que impidieron la adopción de estas herramientas. En el caso de Synapse, la falta de documentación completa y los errores recurrentes en la instalación complicaron su uso. Por su parte, SonataFlow, a pesar de su enfoque prometedor, presentaba problemas de mantenimiento en el repositorio y la imposibilidad de acceder a las imágenes de contenedores Docker necesarias para su despliegue. Estas dificultades técnicas comprometían la viabilidad de su integración en el contexto del trabajo, lo que llevó a descartar su uso a pesar de los beneficios asociados a su alineación con SWS.

2.4.1 Synapse

Synapse es un WFMS de código abierto desarrollado por la comunidad, orientado a la implementación de la SWS. Su propuesta es ofrecer una herramienta flexible que permita a desarrolladores y organizaciones definir y ejecutar *workflows* de manera sencilla utilizando un lenguaje específico del dominio (DSL por sus siglas en inglés) intuitivo y de alto nivel, basado en YAML o JSON [12].

Sin embargo, durante el proceso de evaluación se encontraron obstáculos que limitaron su adopción práctica. La documentación disponible en el repositorio oficial es incompleta y no cubre todos los pasos necesarios para una instalación funcional, lo que obliga al usuario a realizar suposiciones, modificaciones manuales y pruebas por ensayo y error. En particular, se identificaron errores

recurrentes en tiempo de ejecución, como fallos en la descarga de las imágenes Docker utilizadas por el proyecto y falta de claridad sobre la configuración de dependencias internas. Estos problemas, combinados con una baja actividad en el repositorio y escaso soporte comunitario, hacen que cualquier intento de resolver los errores encontrados dependa en gran medida de la intervención directa de los desarrolladores del proyecto, lo cual no siempre es viable.

Dada esta situación, se concluyó que Synapse no ofrece actualmente la madurez ni la estabilidad necesarias para ser utilizado como componente central de este trabajo. Aunque su alineación con un estándar abierto como SWS es positiva, la dificultad para instalarlo y operarlo de forma confiable compromete su viabilidad en un entorno como OSCAR, donde se requiere una herramienta robusta, bien documentada y fácilmente integrable con una plataforma basada en Kubernetes y contenedores. Por estas razones, se decidió descartar su uso en favor de soluciones más maduras y compatibles con los objetivos del proyecto.

2.4.2 SonataFlow

SonataFlow es una iniciativa de código abierto orientada a la implementación de la SWS, cuyo objetivo es facilitar la definición y ejecución de *workflows* en entornos *serverless* mediante una sintaxis declarativa y estandarizada [13]. Al igual que Synapse, SonataFlow busca ofrecer una solución flexible y moderna para la orquestación de tareas sin servidor, con integración en entornos nativos de Kubernetes y soporte para automatización de procesos basada en eventos. En teoría, estas características lo convierten en una alternativa prometedora para su uso junto a plataformas como OSCAR, que también se apoyan en el paradigma de contenedores y ejecución bajo demanda.

Sin embargo, durante el proceso de evaluación, se identificaron problemas relacionados con la sostenibilidad del proyecto y su viabilidad técnica. En particular, el repositorio de código fuente principal de SonataFlow dejó de estar disponible, lo cual genera incertidumbre sobre el mantenimiento futuro del proyecto y la posibilidad de recibir actualizaciones o soporte. Aunque parte de la documentación oficial aún se encuentra accesible, la instalación del sistema no pudo completarse correctamente, debido a fallos en la descarga de las imágenes de contenedor necesarias para su ejecución. Estas imágenes, alojadas en repositorios públicos, no estaban disponibles o arrojaban errores de autenticación o inexistencia, impidiendo el despliegue básico del entorno.

Estas deficiencias comprometen tanto la reproducibilidad como la estabilidad de la herramienta, dos aspectos fundamentales para su incorporación en proyectos de investigación y formación como el presente trabajo. Ante la falta de garantías sobre la continuidad del proyecto y la imposibilidad de realizar pruebas funcionales, se concluyó que SonataFlow no podía ser adoptado como solución viable para la orquestación de *workflows* en OSCAR, a pesar de su alineación conceptual con los principios de la computación *serverless* y los estándares de la CNCF.

2.5 Evaluación de otras herramientas

Además de las herramientas consideradas inicialmente, como Synapse y Apache SonataFlow, se exploraron otras opciones de WFMS que cuentan con amplio reconocimiento en la industria, tales como AWS Step Functions, Apache Airflow, Temporal, Cadence y Prefect. Estas plataformas ofrecen capacidades avanzadas para la definición y ejecución de *workflows*, incluyendo gestión de estado, manejo de errores, reintentos automáticos y monitoreo detallado. Sin embargo, al analizarlas en el contexto específico de integración con OSCAR, se identificaron ciertas limitaciones en términos de compatibilidad, complejidad de despliegue o enfoque de diseño, lo que llevó a descartarlas como solución principal en favor de herramientas más alineadas con arquitecturas basadas en Kubernetes y ejecución *serverless*. A continuación, se detalla el análisis realizado para cada una de estas soluciones.

2.5.1 AWS Step Functions

Se consideró inicialmente la posibilidad de utilizar servicios gestionados en la nube, como los ofrecidos por AWS Step Functions, para la gestión y ejecución de *workflows*. AWS Step Functions es un servicio orquestador de *workflows* basado en estados que permite coordinar múltiples servicios de Amazon Web Services (AWS) de forma visual y declarativa. Utilizando un lenguaje de definición propio llamado Amazon States Language (ASL), los usuarios pueden construir *workflows* compuestos por pasos secuenciales, paralelos, condicionales o con reintentos automáticos, los cuales pueden ejecutar funciones Lambda, llamar a APIs, manipular datos u orquestar tareas distribuidas [14].

Esta plataforma ofrece una solución robusta, escalable y totalmente gestionada, eliminando la necesidad de preocuparse por el aprovisionamiento o mantenimiento de la infraestructura. Además, cuenta con integración de una gran variedad de servicios dentro del ecosistema AWS, como Lambda, DynamoDB, S3, SNS, SQS, y más, lo cual permite construir *workflows* complejos de forma eficiente. Entre sus ventajas se destacan también el monitoreo integrado, el manejo automático de errores, los flujos asíncronos, y su facilidad de uso mediante una interfaz gráfica intuitiva y documentación extensa.

Estas características hicieron que Step Functions se considerara como una alternativa potencialmente atractiva para la orquestación de servicios *serverless* en este trabajo. Sin embargo, su uso fue descartado por motivos prácticos relacionados con su dependencia de la infraestructura de AWS, su modelo de uso basado en cuentas con facturación, y su falta de compatibilidad con los objetivos de autonomía e independencia de plataformas comerciales que se buscan en este proyecto.

El modelo de pago por uso de AWS un factor limitante para este proyecto, cuyo enfoque está orientado al uso de recursos locales o académicos. Al tratarse de un Trabajo de Fin de Máster, se buscó priorizar herramientas de código abierto,

gratuitas y ejecutables en entornos locales o autoalojados, lo que permite mayor control sobre la infraestructura y facilita la reproducción del entorno por parte de terceros sin necesidad de suscripciones o costos adicionales. Por estas razones, se descartó el uso de plataformas comerciales como AWS en favor de soluciones de código abierto compatibles.

2.5.2 Apache Airflow

Apache Airflow es una plataforma de código abierto ampliamente utilizada para la creación, programación y monitoreo de *workflows* definidos como código. Desarrollada originalmente por Airbnb y posteriormente donada a la Apache Software Foundation, Airflow permite modelar *workflows* complejos mediante *scripts* en Python, estructurados como grafos dirigidos acíclicos (Directed Acyclic Graph o DAG por sus siglas en inglés). Cada nodo en el grafo representa una tarea, y las dependencias entre ellas determinan el orden de ejecución. Airflow proporciona una interfaz web para el seguimiento de las ejecuciones, herramientas para reintentos automáticos, control de dependencias, y soporte para tareas periódicas (*cron jobs*) [15]. Gracias a su flexibilidad y extensibilidad, se ha convertido en una herramienta popular en entornos de ciencia de datos, ingeniería de datos y automatización de pipelines.

Aunque Apache Airflow es una de las herramientas más consolidadas y utilizadas para la orquestación de *workflows* en entornos de procesamiento de datos, su adopción en un entorno como OSCAR presenta varias limitaciones.

Primero, Airflow no fue diseñado originalmente para ejecutarse en Kubernetes ni para coordinar la ejecución de tareas como contenedores efímeros, que es precisamente el modelo de ejecución que utiliza OSCAR. Aunque existen integraciones como el *KubernetesExecutor* o el *KubernetesPodOperator*, estas añaden una capa de complejidad adicional y no ofrecen una integración tan nativa y directa como la que proporciona Argo Workflows, que fue construido desde cero para Kubernetes.

En segundo lugar, Airflow tiene una arquitectura monolítica y más pesada, que incluye múltiples componentes que deben mantenerse activos: base de datos, servidor web, *scheduler*, *worker(s)*, y a veces Redis o Celery si se usa ejecución distribuida [16]. Este enfoque aumenta los requisitos de infraestructura y mantenimiento, algo que puede resultar poco práctico en entornos académicos o de investigación donde se prioriza la sencillez y el despliegue rápido.

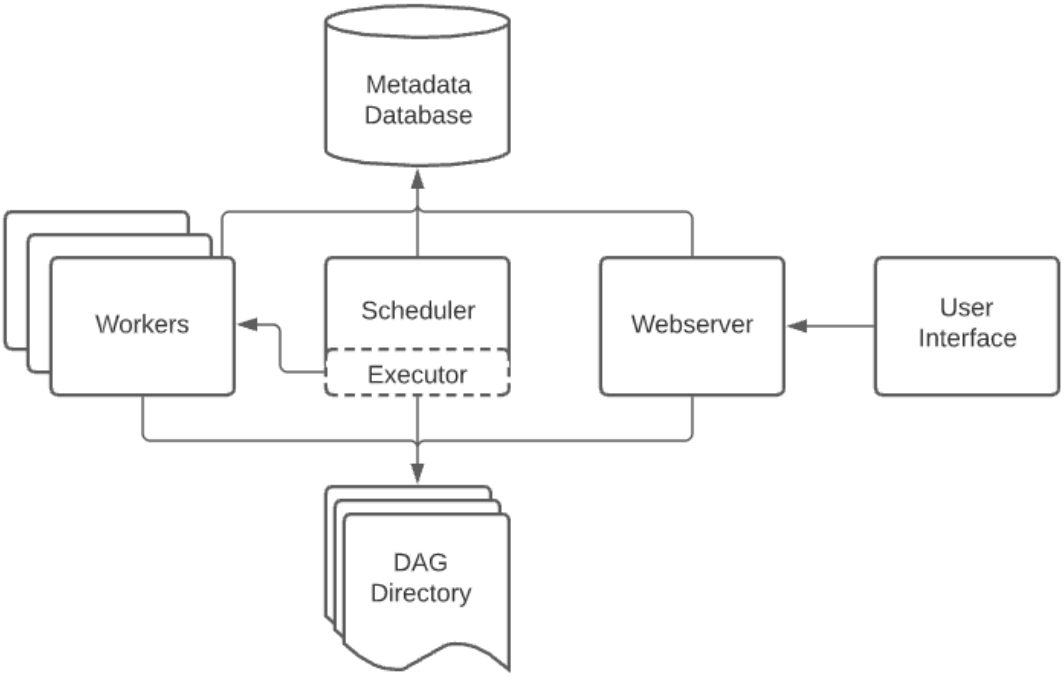


Figura 2: Arquitectura básica de Apache Airflow, fuente [16]

Por último, aunque Airflow permite definir tareas mediante código Python, sus *workflows* no están pensados para ser ejecutados en entornos puramente *serverless*. Cada tarea suele estar acoplada a un entorno persistente de ejecución y no se alinea fácilmente con el modelo de servicios efímeros activados por eventos que propone OSCAR.

2.5.3 Temporal

Temporal [16] es una plataforma *open-source* de orquestación de *workflows* que permite a los desarrolladores definir y ejecutar procesos complejos de larga duración a través de código. A diferencia de otras herramientas que utilizan descripciones declarativas (como YAML), Temporal permite modelar *workflows* como funciones en lenguajes de programación como Go, Java o TypeScript, ofreciendo flexibilidad en la lógica de ejecución, manejo de errores, condiciones, y paralelismo. Durante la fase de evaluación de tecnologías, Temporal se consideró como una alternativa atractiva debido a su fiabilidad, escalabilidad y su enfoque moderno en la orquestación distribuida.

Uno de los principales puntos fuertes de Temporal es su capacidad de persistencia automática del estado del *workflow*: si un proceso se interrumpe por un error, reinicio o fallo en la infraestructura, puede continuar exactamente desde donde se detuvo sin necesidad de redefinir la lógica de recuperación. Además, Temporal ofrece manejo integrado de reintentos, *timeouts*, ejecución paralela, cancelación y versionado de *workflows*, todo directamente desde el código. Esta arquitectura lo convierte en una solución potente para sistemas que requieren alta disponibilidad y consistencia en procesos complejos.

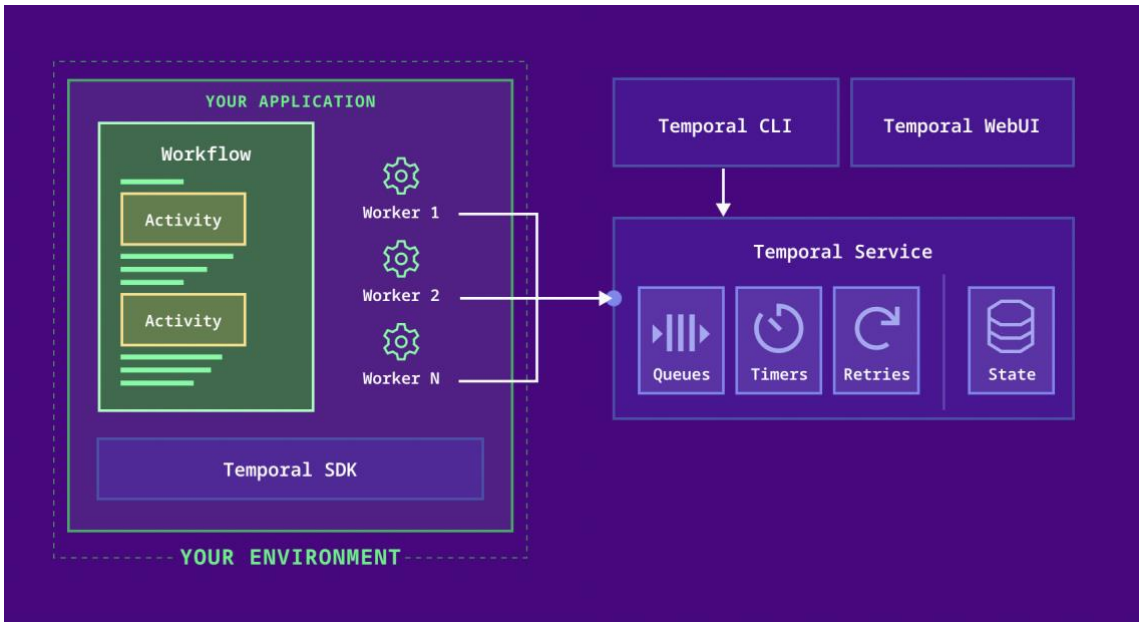


Figura 3: Máquina de estado en Temporal, fuente [17]

Sin embargo, Temporal también presenta desventajas importantes en el contexto específico de este trabajo. Su despliegue implica configurar múltiples servicios (*frontend*, *elastic-search*, base de datos, etc.), lo cual introduce una mayor complejidad operativa [17]. Así mismo, para interactuar con el servidor de Temporal y definir los *workflows*, es necesario dominar la SDK de Temporal y familiarizarse con los conceptos específicos de esta herramienta, lo que supone una curva de aprendizaje más elevada sobre un dominio específico solo para esta herramienta.

Por último, la arquitectura de Temporal no está centrada en Kubernetes ni en la ejecución de contenedores de forma directa. En Temporal, los *workflows* se definen como funciones escritas en código y ejecutadas por procesos llamados *workers*, que deben estar siempre activos y conectados al servidor. Este modelo contrasta con el enfoque de OSCAR, donde los servicios se ejecutan de forma provisional y aislada en contenedores, sin necesidad de mantener procesos residentes. Por lo tanto, la ejecución basada en *workers* no se alinea con el modelo *serverless* por contenedor que caracteriza a OSCAR.

2.5.4 Cadence

Cadence es una plataforma de orquestación de *workflows* distribuida diseñada para simplificar el desarrollo de aplicaciones complejas y resilientes. Surge como solución a la necesidad de gestionar *workflows* que van más allá de simples interacciones solicitud-respuesta, involucrando múltiples pasos, eventos asíncronos y dependencias externas poco fiables. En lugar de recurrir a múltiples sistemas desconectados como servicios sin estado, tareas periódicas o colas de mensajes, Cadence ofrece un modelo de programación con estado y tolerante a fallos, que permite escribir *workflows* directamente en lenguajes como Go o Java, mientras la plataforma garantiza la durabilidad, escalabilidad y disponibilidad

del flujo completo. Su arquitectura se compone de un *framework* cliente y un *backend* gestionado, el cual es *stateless* y persiste el estado de los *workflows* en bases de datos como Cassandra o MySQL/PostgreSQL. Esta combinación permite a los desarrolladores centrarse en la lógica de negocio sin preocuparse por la complejidad operativa habitual de los sistemas distribuidos [18].

A pesar de sus capacidades avanzadas y su enfoque robusto para la orquestación de *workflows* distribuidos, Cadence presenta limitaciones significativas al considerarse para su integración con la plataforma OSCAR, la mayoría son las mismas que presenta Temporal. Una de las principales barreras radica en su arquitectura, la cual está compuesta por múltiples servicios internos (incluyendo *frontend*, *history*, *matching* y *worker services*) además de requerir una base de datos persistente como Cassandra o MySQL. Esta infraestructura compleja no está diseñada para entornos basados en ejecución efímera de contenedores, como es el caso de OSCAR, donde los servicios deben activarse de forma puntual y sin la necesidad de mantener procesos en ejecución permanente.

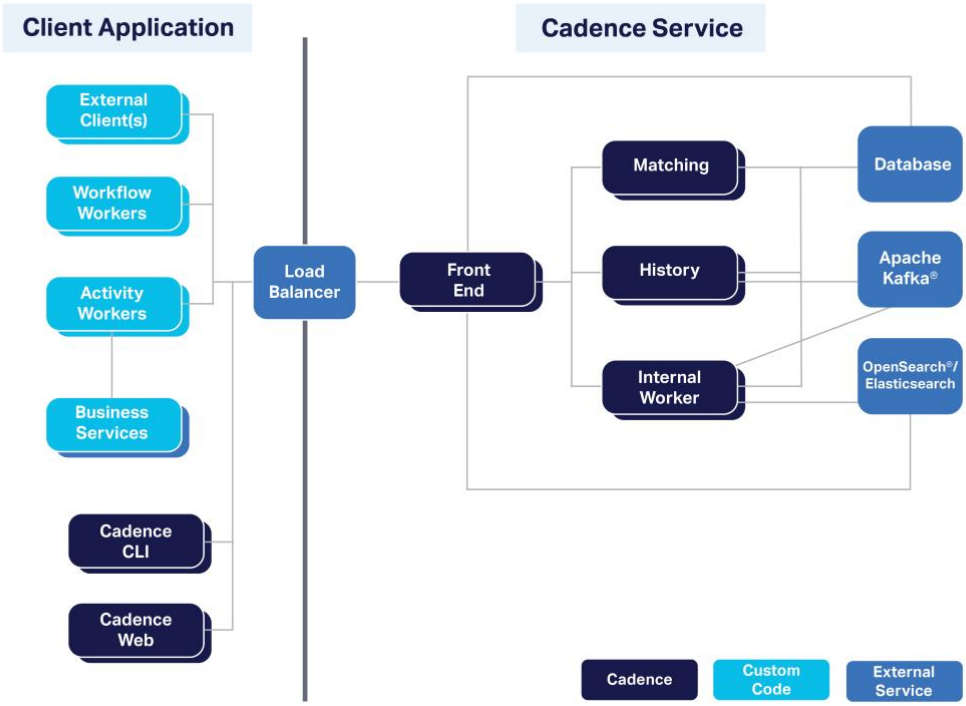


Figura 4: Arquitectura de un servicio Cadence, fuente [19]

Otro factor importante es que Cadence requiere que los *workflows* sean definidos como código, lo cual implica que los usuarios deben compilar y ejecutar *workers* dedicados que se mantengan conectados al *backend* para procesar tareas. Este enfoque entra en conflicto con el modelo de OSCAR, en el que los servicios se ejecutan de manera aislada, reproducible y bajo demanda como contenedores autocontenidos. Además, la necesidad de mantener *workers* activos en todo momento dificulta la escalabilidad en entornos académicos o de investigación, donde los recursos son limitados y se prioriza la simplicidad en el despliegue. En

conjunto, estas características hacen que Cadence no se alinee adecuadamente con los principios de diseño de OSCAR, motivo por el cual fue descartado como opción viable para este trabajo.

2.5.5 Prefect

Prefect es un motor de orquestación de *workflows* de código abierto que permite transformar funciones escritas en Python en pipelines de datos robustos y listos para producción. A diferencia de herramientas que requieren lenguajes específicos o archivos de configuración complejos, Prefect permite definir *workflows* utilizando Python puro, lo que facilita su integración en entornos de desarrollo modernos. Ofrece funcionalidades como gestión automática de estados, manejo de errores, programación flexible y monitoreo en tiempo real, todo mediante una sintaxis familiar para desarrolladores. Desde su aparición en 2018, Prefect ha ganado popularidad como una alternativa moderna a Airflow, adaptada a las necesidades de *workflows* dinámicos y entornos de ejecución heterogéneos [20].

A pesar de su diseño moderno y su orientación a *workflows* definidos completamente en Python, Prefect presenta limitaciones similares a las ya expuestas en las evaluaciones de Temporal y Cadence, particularmente en lo que respecta a su necesidad de mantener procesos residentes (*agents* o *workers*) conectados a un servidor orquestador. Este modelo operativo no se ajusta al enfoque sin estado y basado en contenedores efímeros que define a OSCAR, y, por tanto, ya supone una barrera de entrada para su integración.

Además de este desacoplamiento, Prefect introduce otras incompatibilidades. Por ejemplo, muchas de sus funcionalidades clave (como la monitorización en tiempo real, la visualización de dependencias o el uso de su interfaz gráfica) dependen del Prefect Cloud o del Prefect Server, lo que añade una capa de infraestructura adicional que no se alinea con los principios de despliegue simplificado y autosuficiencia que se buscan en entornos como OSCAR. También, dado que Prefect no opera nativamente con un enfoque declarativo como Argo Workflows, su integración en flujos GitOps, control de versiones o ejecución reproducible de pipelines en YAML es limitada.

2.6 Argo Workflows

Una vez considerados los distintos enfoques y plataformas disponibles para la gestión de *workflows* en entornos *serverless*, se optó por utilizar Argo Workflows como la herramienta principal de este trabajo. Argo Workflows es una herramienta de código abierto diseñada específicamente para la definición y ejecución de *workflows* sobre clústeres de Kubernetes [3]. Su enfoque declarativo, su estrecha integración con contenedores y su arquitectura basada en recursos nativos de Kubernetes la convierten en una opción idónea para entornos como OSCAR, donde se busca automatizar tareas distribuidas sin recurrir a infraestructura persistente. A continuación, se describe en detalle su

funcionamiento, componentes principales y las razones que justifican su adopción para este proyecto.

2.6.1 Arquitectura general

Argo Workflows está construido para ejecutarse sobre clústeres de Kubernetes, aprovechando su modelo de operación basado en contenedores. Su arquitectura se compone de varios componentes clave que permiten la definición, ejecución y monitoreo de *workflows* de forma eficiente y escalable.

El componente principal es el *Workflow Controller*, un operador de Kubernetes que observa los recursos de tipo *Workflow* y se encarga de coordinar su ejecución creando los *pods* necesarios en el orden indicado. A medida que el flujo avanza, el controlador actualiza el estado del *workflow* y gestiona aspectos como reintentos, fallos o condiciones de salida.

Otro componente importante es el *Argo Server*, un servicio opcional que expone una API REST y una interfaz gráfica de usuario (UI por sus siglas en inglés). Este servidor facilita la interacción con los *workflows* en ejecución o finalizados y permite gestionar recursos como plantillas, *workflows* programados o parámetros, todo desde una interfaz web fácil de usar.

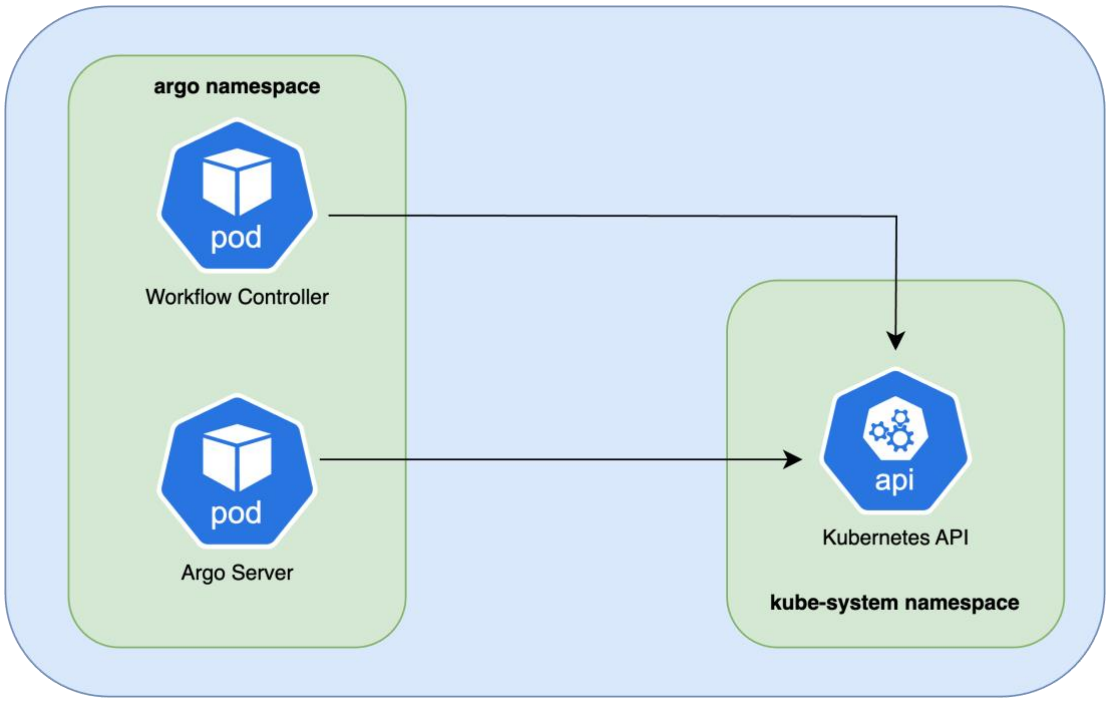


Figura 5: Componentes principales de Argo Workflows

El esquema recogido en la Figura 5 resume de forma clara los componentes fundamentales de Argo Workflows y su interacción con Kubernetes para la ejecución de tareas. Sin embargo, es importante destacar que Argo Workflows puede formar parte de un ecosistema más amplio de orquestación, donde diversas herramientas e integraciones complementen su funcionalidad. Por

ejemplo, Argo Events permite activar *workflows* en respuesta a eventos externos como mensajes en colas (Kafka, NATS), cambios en *buckets* de almacenamiento (S3, MinIO), o solicitudes HTTP. Para gestionar entradas y salidas de los *workflows*, es común integrar sistemas de almacenamiento de objetos como Amazon S3, Google Cloud Storage, MinIO, o incluso volúmenes persistentes de Kubernetes. Además, Argo Workflows puede integrarse con sistemas de registro y monitoreo como Prometheus y Grafana para visualizar métricas y estados de ejecución, así como con soluciones de autenticación como OIDC, RBAC de Kubernetes, o autenticación por *tokens*. También es posible incorporar Argo CD (Argo Continuous Deployment) para facilitar el despliegue continuo de *workflows* definidos como código. En conjunto, estas herramientas permiten construir pipelines robustos, reproducibles y adaptables a diferentes entornos y necesidades operativas.

2.6.2 Ventajas de utilizar Argo Workflows en OSCAR

Madurez y comunidad activa
Argo Workflows cuenta con una comunidad de desarrollo activa y un repositorio con alta actividad en GitHub, lo que garantiza soporte y mejoras constantes en la plataforma [21].

Documentación detallada
Su documentación es muy completa, facilitando su instalación y configuración sin necesidad de realizar modificaciones adicionales y poniendo a disponibilidad del público una referencia clara y detallada sobre las funcionalidades y formas de diseñar los *workflows*. En el momento de redacción de este documento, también está disponible un curso de acceso gratuito que permite aprender el uso de la herramienta sin necesidad de instalar recursos localmente [23].

Integración nativa con Kubernetes
Dado que Argo Workflows está diseñado específicamente para ejecutarse en Kubernetes, puede aprovechar las capacidades de escalabilidad y gestión de recursos de esta plataforma, facilitando la ejecución eficiente de *workflows serverless*.

Compatibilidad con OSCAR
Argo Workflows, al estar basado en Kubernetes, es completamente compatible con la arquitectura de OSCAR, permitiendo la orquestación eficiente de *workflows* dentro de estos clústeres. La naturaleza basada en contenedores de ambos sistemas facilita el aislamiento y la reproducibilidad de tareas, asegurando que las funciones *serverless* se ejecuten de manera controlada y optimizada. Además, al soportar ejecución distribuida y paralelización de tareas, Argo Workflows encaja perfectamente en la visión de OSCAR de crear aplicaciones *serverless* altamente escalables en la nube y en infraestructuras *on-premises*.

Soporte para múltiples entornos
Argo Workflows permite la orquestación de tareas en diferentes entornos, incluyendo servidores *on-premises* y servicios *cloud* como AWS Lambda, proporcionando una solución versátil para la ejecución de *workflows* híbridos.

2.7 Conclusión

En resumen, la revisión de herramientas de orquestación de *workflows* permitió identificar diversas alternativas robustas y ampliamente utilizadas en la industria. Cada una de estas soluciones ofrece características valiosas y enfoques distintos para la gestión de *workflows*, desde modelos declarativos hasta programación completamente basada en código. No obstante, al analizar su aplicabilidad en el contexto específico de este trabajo, centrado en la integración con la plataforma OSCAR, se identificaron ciertas limitaciones en términos de compatibilidad de arquitectura, dependencia de procesos persistentes o complejidad en el despliegue. Es importante destacar que estas observaciones no invalidan el valor ni la utilidad de estas herramientas en otros escenarios, donde sus características pueden resultar altamente beneficiosas. Por el contrario, esta evaluación permitió seleccionar la herramienta que mejor se adapta a las particularidades del entorno objetivo: Argo Workflows, una solución ligera, declarativa y diseñada desde su origen para ejecutarse en Kubernetes, lo que la convierte en una opción especialmente adecuada para el tipo de orquestación basada en contenedores que propone OSCAR.

3. Desarrollo

Una vez seleccionada la herramienta de orquestación más adecuada para el entorno objetivo, este apartado se centra en describir el proceso de implementación práctica de Argo Workflows en un clúster OSCAR. Se abordan aspectos técnicos y operativos relacionados con su despliegue, configuración, integración con los servicios existentes y validación mediante ejemplos funcionales. El objetivo es demostrar la viabilidad de utilizar Argo para coordinar workflows en un entorno *serverless* distribuido, destacando su capacidad de automatización, modularidad y facilidad de uso en contextos académicos o de investigación. A lo largo de este capítulo se detallan tanto los componentes del entorno como las decisiones de diseño adoptadas, las plantillas utilizadas y los flujos implementados para distintos casos de uso.

3.1 Entorno de trabajo

El desarrollo de este proyecto se llevó a cabo en un entorno de pruebas controlado, basado en un clúster OSCAR donde se instaló Argo Workflows como sistema de orquestación. Esta sección describe los componentes técnicos que conforman dicho entorno, así como las ventajas de ejecutar Argo y OSCAR en un mismo clúster Kubernetes. Se examinan aspectos clave como la comunicación entre servicios, la integración con sistemas de almacenamiento (MinIO), la gestión de autenticación y el aprovechamiento de recursos compartidos. También se presentan herramientas auxiliares como el *dashboard* de OSCAR y los servicios de automatización, con el fin de contextualizar la infraestructura sobre la que se construyeron y validaron los workflows desarrollados en este trabajo.

3.1.1 Descripción del entorno utilizado

Para este proyecto se utilizó un clúster OSCAR de prueba, en el que también se instaló Argo Workflows. Gracias a esto, los *workflows* se ejecutan directamente dentro del mismo entorno en el que están los servicios, lo que facilita las pruebas y el desarrollo.

La instalación de Argo Workflows directamente dentro del clúster de OSCAR ha ofrecido importantes ventajas tanto técnicas como prácticas. Una de las principales ventajas fue la posibilidad de establecer una comunicación directa entre Argo Workflows y los servicios de OSCAR. Al encontrarse dentro del mismo clúster de Kubernetes, los componentes pueden comunicarse internamente sin necesidad de exponer servicios hacia el exterior ni configurar reglas de red o autenticación adicionales más allá de los existentes en OSCAR. Esto facilita notablemente la integración entre ambos sistemas, ya que los *workflows* pueden invocar servicios directamente utilizando rutas internas del clúster.

Además, esta configuración reduce considerablemente la complejidad de la instalación y puesta en marcha. No es necesario configurar certificados, dominios personalizados ni servicios intermedios para facilitar la comunicación entre los componentes de Argo Workflows y OSCAR; en su lugar, se aprovechan las funcionalidades nativas del entorno Kubernetes, lo que contribuye a una configuración más sencilla y robusta. También se observa un mejor rendimiento, ya que todas las comunicaciones entre los componentes ocurren dentro de la red del clúster, eliminando la latencia que podría introducirse al depender de servicios externos o de una infraestructura en la nube.

Desde el punto de vista del desarrollo y la depuración, tener Argo Workflows y OSCAR instalados en el mismo entorno local ha facilitado enormemente la experimentación. El acceso directo a los registros de ejecución y a los recursos del sistema permite iterar rápidamente en el diseño de los *workflows*, detectar errores con mayor facilidad y probar distintas configuraciones sin restricciones externas.

Esta integración también mejora la portabilidad y la reproducibilidad del entorno. Al estar todo contenido en un único clúster, es posible compartir o replicar la configuración completa en otras máquinas lo cual es especialmente valioso en entornos académicos o colaborativos. Además, al compartir el mismo entorno, tanto Argo Workflows como OSCAR pueden acceder sin problemas a recursos como volúmenes persistentes, secretos de Kubernetes o servicios como MinIO, lo que permite construir *workflows* más completos y seguros aprovechando las capacidades integradas del clúster.

3.1.2 Otras arquitecturas posibles

Al ser Argo Workflows una herramienta ligera y fácil de instalar en cualquier entorno que cuente con Kubernetes, también resulta viable su despliegue en un entorno separado del clúster donde residen los servicios que ejecutan las tareas. Esta capacidad abre la puerta a arquitecturas más flexibles, como por ejemplo el uso de Argo Workflows como orquestador remoto, desde otro clúster o dispositivo, lo que puede aportar varios beneficios relevantes en escenarios avanzados.

Entre los beneficios potenciales se encuentra la posibilidad de implementar modelos de infraestructura multi-nube o híbrida, en los cuales Argo Workflows actúa como una capa de control centralizada que coordina servicios desplegados en distintas nubes o centros de datos. Esto es especialmente útil en organizaciones que deben distribuir sus cargas de trabajo entre distintos proveedores por razones de costo, latencia, soberanía de datos o disponibilidad.

Además, este enfoque puede contribuir a un modelo de computación continua o computación federada, donde el procesamiento se reparte entre entornos locales (por ejemplo, en el borde o en universidades) y clústeres remotos en la nube. Argo Workflows puede encargarse de orquestar estos flujos distribuidos, gestionando

la transferencia de datos entre entornos y el encadenamiento de tareas, sin necesidad de que todos los recursos estén en un mismo clúster.

Otro beneficio importante es el desacoplamiento entre la lógica de orquestación y la infraestructura de ejecución, lo cual puede simplificar el mantenimiento, la escalabilidad y la seguridad. Por ejemplo, un entorno centralizado donde reside Argo Workflows podría ser gestionado por un equipo diferente al que administra los clústeres de ejecución, permitiendo una separación clara de responsabilidades y facilitando el cumplimiento de políticas de la organización.

3.2 Instalación de Argo Workflows en OSCAR

Para la instalación de Argo Workflows y su herramienta de línea de comandos argo-cli [39], se siguió la documentación oficial [40]. El proceso de instalación de Argo Workflows es directo y está diseñado para integrarse de forma sencilla con cualquier entorno que disponga de un clúster de Kubernetes funcional. La instalación comienza con la preparación del entorno, lo cual implica disponer de acceso administrativo a un clúster de Kubernetes y tener configurada la herramienta de línea de comandos *kubectl* para comunicarse con el mismo.

Una vez configurado el entorno, se procede a desplegar Argo Workflows en el clúster mediante la aplicación de un manifiesto YAML que se encuentra disponible en el repositorio oficial del proyecto. Este manifiesto contiene la definición de los recursos necesarios, como el controlador de *workflows* (Workflow Controller), el servidor de Argo Workflows (Argo Server) y las definiciones de recursos personalizados que permiten al clúster reconocer y gestionar objetos del tipo Workflow o WorkflowTemplate.

Durante este proyecto, se instaló la versión v3.6.7 de Argo Workflows, disponible en el repositorio oficial del proyecto en GitHub [30]. Al instalarse en un espacio de nombre (*namespace*) [33] específico llamado *argo*, se logró mantener una separación clara entre los recursos propios de Argo Workflows (como sus controladores, servicios y configuraciones) y los del resto del clúster. Esta organización facilita la gestión del sistema, permite aplicar reglas de acceso más precisas, y mejora la visibilidad de los componentes durante la depuración o el monitoreo. Además, al utilizar un *namespace* dedicado, es posible reinstalar o actualizar Argo Workflows sin afectar a otros servicios desplegados en el mismo clúster.

En la Figura 6 se ilustra la interacción que puede establecer un usuario con el sistema OSCAR a través de Argo Workflows, ya sea utilizando la interfaz gráfica de Argo Workflows (expuesta como servicio) o accediendo directamente mediante su API REST. A través de estos medios, el usuario tiene la capacidad de definir, enviar y gestionar *workflows* que orquestan la ejecución de servicios OSCAR. Estos servicios pueden activarse de diferentes maneras: mediante la carga de archivos a un *bucket* de MinIO (lo que desencadena la ejecución

automática de servicios que reaccionan a eventos de almacenamiento) o mediante llamadas directas a la API de OSCAR.

Una vez que el *workflow* ha sido creado, el Workflow Controller de Argo Workflows se encarga de interpretar su definición declarativa y coordinar la ejecución de cada uno de sus pasos. Esta coordinación se realiza comunicándose directamente con la API de Kubernetes, creando *pods* según lo especificado y gestionando su ciclo de vida, incluyendo aspectos como dependencias, reintentos, salidas, errores y finalización del flujo. De esta manera, Argo Workflows actúa como puente entre el usuario y la infraestructura subyacente, permitiendo una orquestación flexible y automatizada de servicios dentro del ecosistema OSCAR.

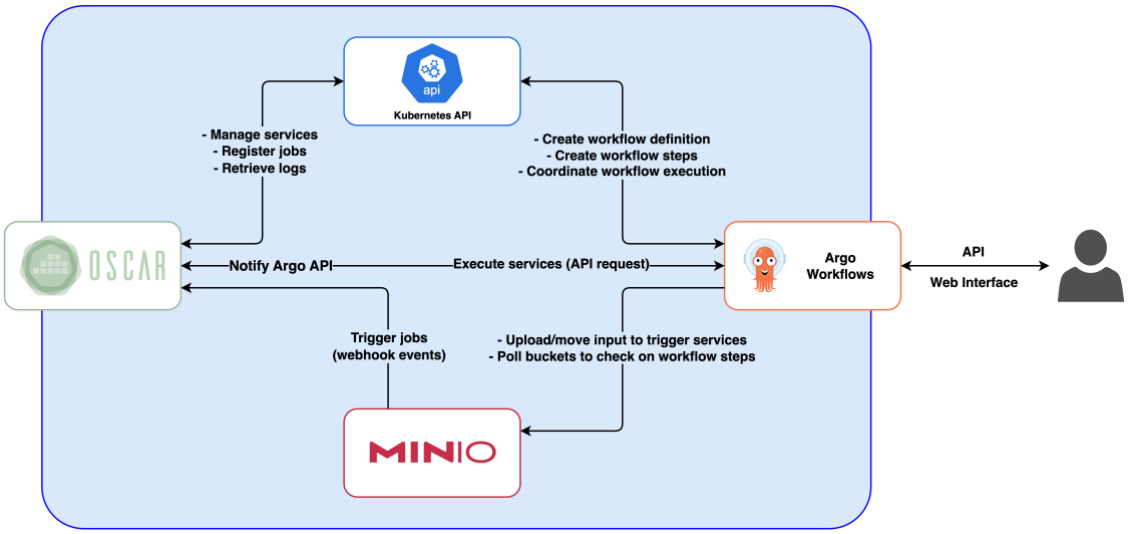


Figura 6: Interacción de la ejecución de un workflow en OSCAR

3.2.1 Argo User Interface

La interfaz de usuario de Argo Workflows es una aplicación web que permite visualizar, ejecutar y gestionar *workflows* de forma gráfica. A través de esta interfaz, es posible seguir en tiempo real el estado de cada paso del *workflow*, revisar los parámetros de entrada y salida, consultar los registros de ejecución, y lanzar nuevas ejecuciones sin necesidad de usar la línea de comandos. Es una herramienta muy útil para depuración, monitoreo y control visual de los *workflows*.

Acceso al servicio

La interfaz de usuario de Argo Workflows no está expuesta por defecto principalmente por razones de seguridad y flexibilidad de despliegue. Al dejar la interfaz sin exponer públicamente, se evita que cualquier usuario externo pueda acceder al sistema de orquestación, que suele tener permisos elevados para ejecutar tareas, leer secretos o modificar recursos del clúster. Esto es especialmente importante en entornos compartidos o de producción, donde una interfaz mal protegida podría representar una vulnerabilidad crítica.

Para hacer accesible la interfaz gráfica de Argo Workflows, se definió un recurso Ingress (Figura 7) que permite acceder al servicio web desde fuera del clúster [34], sustituyendo el uso de *kubectl port-forward*, el cual limita el acceso a la máquina local y requiere mantener una terminal activa. Esta configuración permite asociar un dominio personalizado (como argo.competent-sanderson0.im.grycap.net) y habilitar el uso de HTTPS mediante un certificado TLS. Dicho certificado es emitido automáticamente por Let's Encrypt a través de Cert-Manager y almacenado en un secreto TLS, el cual es referenciado en la definición del Ingress. Gracias a esta arquitectura, es posible acceder de forma persistente y segura a la interfaz de Argo Workflows desde cualquier navegador web.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: argo-server
  namespace: argo
  annotations:
    nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    cert-manager.io/cluster-issuer: letsencrypt-prod
  labels:
    app: argo-workflows
spec:
  ingressClassName: nginx
  rules:
    - host: argo.competent-sanderson0.im.grycap.net
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: argo-server
                port:
                  number: 2746
  tls:
    - hosts:
        - argo.competent-sanderson0.im.grycap.net
      secretName: argo.competent-sanderson0.im.grycap.net
```

Figura 7: Manifiesto del Ingress que permite visualizar la interfaz gráfica de Argo Workflows

Token de acceso

Para este proyecto se optó por utilizar autenticación mediante *token* en lugar de un sistema más complejo como inicio de sesión único o *Single Sign-On* (SSO) en inglés. Esta decisión se tomó en coherencia con el modelo de autenticación utilizado por el despliegue de prueba de OSCAR, que se basa en credenciales simples (nombre de usuario y contraseña), si bien OSCAR también soporta integración con otros sistemas de autenticación basados en OIDC como EGI Check-In o Keycloak. Al utilizar *tokens* estáticos asociados a cuentas de servicio, se mantiene una línea de acceso sencillo y directo, sin necesidad de integrar servicios externos de identidad o configuración adicional.

```
kubectl create sa argo-user -n argo
```

Figura 8: Comando utilizado para crear la cuenta de servicio argo-user

Para ello, se creó una cuenta de servicio llamada *argo-user* en el *namespace argo* (Figura 8), la cual se asoció a un *ClusterRole* personalizado con los permisos necesarios (Figura 9) para listar, crear, actualizar y eliminar *workflows* y plantillas, así como obtener, enlistar y observar *pods*; permisos necesarios para poder observar los logs y el progreso de los *workflows* correctamente. Finalmente, se estableció un *ClusterRoleBinding* (Figura 10) que enlaza dicha cuenta de servicio con el rol creado, permitiendo así su uso a nivel de clúster. Es importante destacar que existen otros recursos dentro de Argo Workflows a los que esta cuenta de servicio no tendrá acceso. Sin embargo, dado que dichos recursos no serán utilizados en el contexto de este proyecto, lo más recomendable es no otorgar permisos innecesarios, siguiendo así el principio de privilegio mínimo. El principio del privilegio mínimo especifica que cada programa y cada usuario del sistema debe operar con el mínimo conjunto de privilegios necesario para completar su trabajo. Principalmente, este principio limita el daño que puede resultar de un accidente o error. También reduce al mínimo el número de interacciones potenciales entre programas privilegiados para un correcto funcionamiento, de modo que sea menos probable que se produzcan usos involuntarios, no deseados o indebidos de los privilegios [35].

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: argo-user-role
rules:
  - apiGroups: ["argoproj.io"]
    resources: ["workflowtemplates", "workflows"]
```

```
    verbs: ["get", "list", "watch", "create", "update", "delete"]
  - apiGroups: [""]
    resources: ["pods", "pods/log"]
    verbs: ["get", "list", "watch", "delete"]
```

Figura 9: Manifiesto del rol que permitirá a argo-user administrar workflows

```
kubectl create clusterrolebinding argo-user-binding --clusterrole=argo-user-role --serviceaccount=argo:argo-user
```

Figura 10: Crear binding del rol y la cuenta de servicio

Para obtener el *token* de autenticación, se creó un secreto para asociarlo a la cuenta de servicio previamente creada (Figura 11). Una vez aplicado el manifiesto, el *token* fue extraído utilizando un comando *kubectl*, decodificando su contenido en base64 (Figura 12). Este *token* puede ser utilizado para autenticarse en la interfaz web de Argo Workflows (insertando la palabra "Bearer" previo al *token*), copiándolo en el campo correspondiente al iniciar sesión (Figura 13), o para realizar llamadas HTTP a la API de Argo Workflows (Figura 14).

```
apiVersion: v1
kind: Secret
metadata:
  name: argo-user.service-account-token
  namespace: argo
  annotations:
    kubernetes.io/service-account.name: argo-user
type: kubernetes.io/service-account-token
```

Figura 11: Secreto que albergará el token de la cuenta

```
kubectl -n argo get secret argo-user.service-account-token -o jsonpath="{.data.token}" | base64 -d
```

Figura 12: Comando para obtener el token del secreto previamente creado

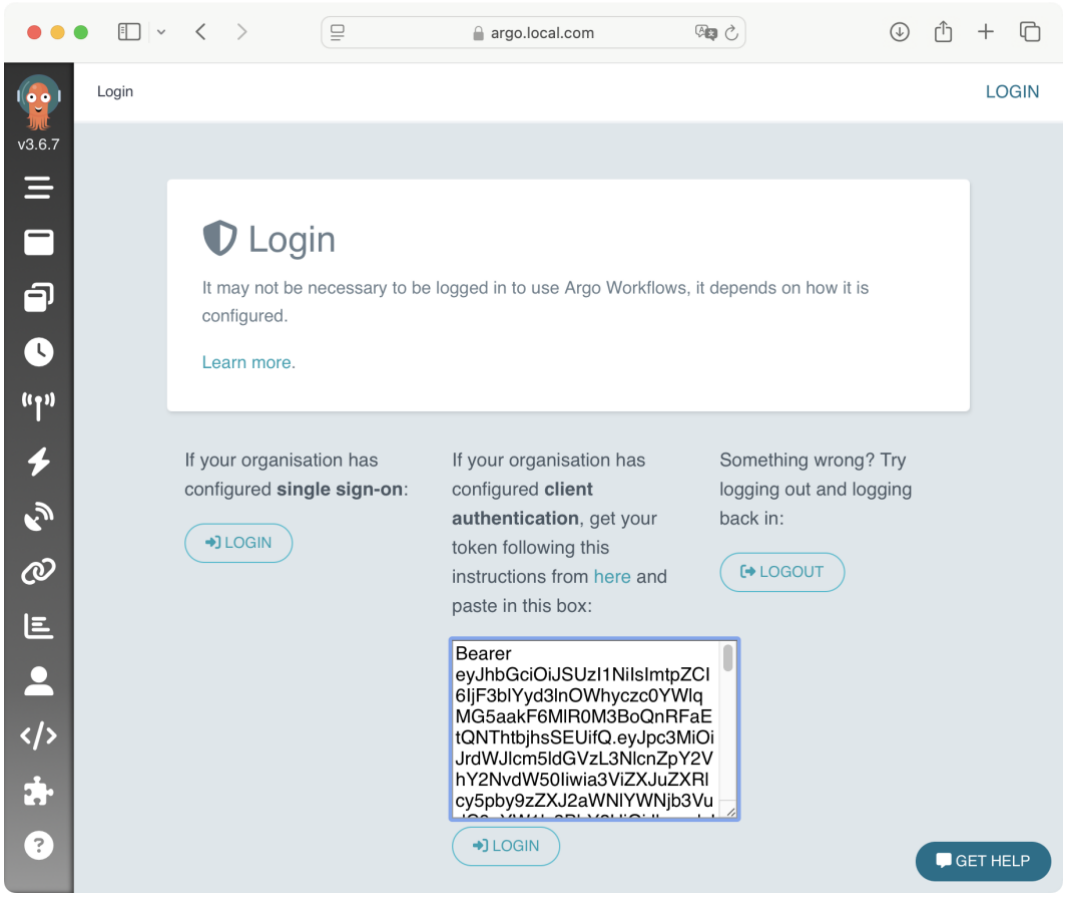


Figura 13: Pantalla de inicio de la interfaz gráfica de Argo Workflows

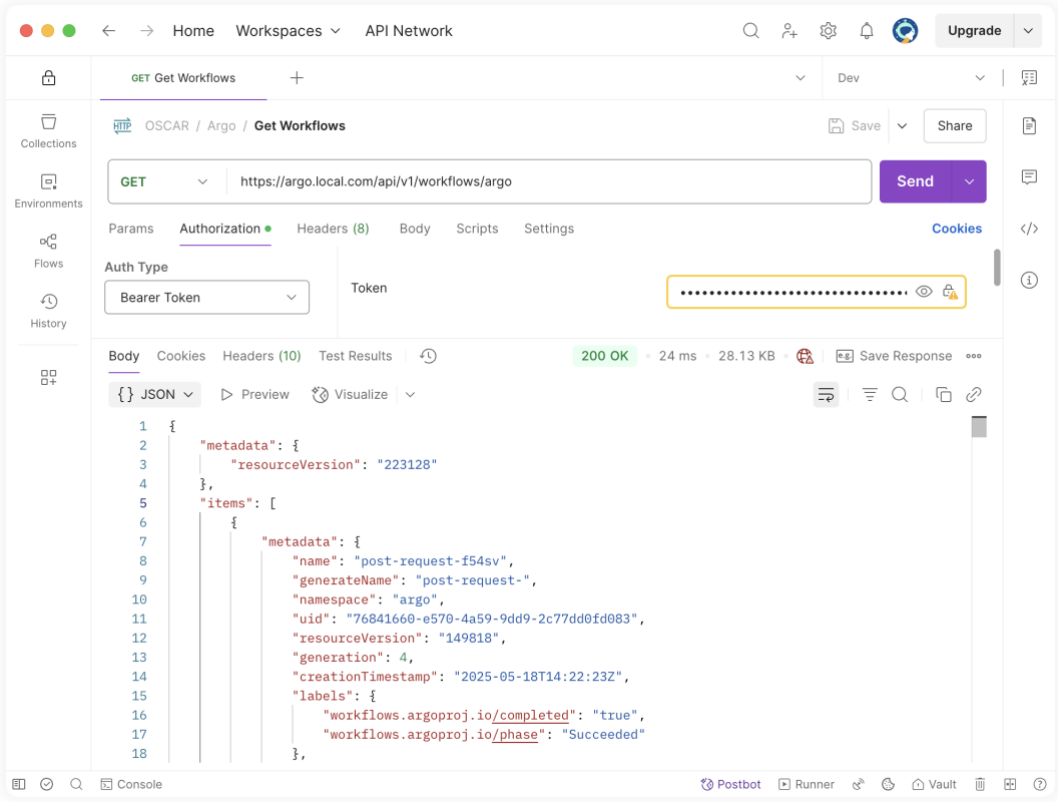


Figura 14: Solicitud HTTP a la API de Argo Workflows

Acceso mediante OAuth

Un clúster OSCAR se puede implementar fácilmente en múltiples plataformas de nube mediante el Panel de Control del Gestor de Infraestructura (IM Dashboard). Este es un servicio gestionado por EGI y operado por el grupo de investigación GRyCAP de la Universitat Politècnica de València para implementar infraestructuras virtuales personalizadas en múltiples proveedores de nube [41].

Así mismo, OSCAR puede integrar EGI Check-in [42], una solución proporcionada por la European Grid Infrastructure (EGI) que permite a los usuarios acceder a servicios distribuidos utilizando sus credenciales institucionales. EGI Check-in actúa como un proveedor de identidad (IdP) federado, compatible con estándares como OpenID Connect (OIDC) y SAML, y permite a los servicios como OSCAR delegar la gestión de autenticación. Entre sus ventajas destacan el SSO, la compatibilidad con múltiples fuentes de identidad (como universidades, ORCID o proveedores sociales), y la asignación automática de roles basada en atributos del usuario.

Argo Workflows es compatible con autenticación mediante OIDC, lo que lo hace perfectamente integrable con entornos como OSCAR integrados con EGI Check-in. En este tipo de configuraciones, Argo Workflows puede delegar la autenticación a EGI Check-in y aplicar políticas de control de acceso mediante RBAC según los atributos del usuario autenticado. Esta integración es útil en contextos con múltiples usuarios, donde se requiere una gestión centralizada, trazabilidad y cumplimiento de políticas de seguridad establecidas por las instituciones participantes.

Este apartado se incluyó únicamente para evidenciar que Argo Workflows es compatible con métodos avanzados de autenticación. Sin embargo, esta funcionalidad no se implementó ni se probó en la práctica, ya que el clúster de prueba no contaba con un sistema de autenticación basado en OAuth. Por lo tanto, el análisis se limita a una explicación teórica, sin una validación técnica en el entorno de pruebas.

3.4 Integración

Este apartado está dedicado a describir el proceso de integración entre Argo Workflows y los servicios desplegados en OSCAR, centrándose particularmente en los mecanismos de activación de dichos servicios. Para ello, se exploran dos enfoques principales: la ejecución a través de llamadas directas a la API de OSCAR y la activación mediante la carga de archivos en *buckets* gestionados por MinIO. Estos mecanismos permiten incorporar servicios de forma flexible dentro de *workflows* más amplios, facilitando la automatización de tareas de análisis, inferencia o procesamiento de datos. A lo largo de esta sección se presentarán fragmentos representativos del código y de los manifiestos YAML necesarios para implementar ambos enfoques, mientras que los ejemplos completos estarán disponibles en el repositorio público *smill3r/OSCAR-Argo-Workflows* [36], con el fin de facilitar su consulta, réplica o adaptación.

3.3.1 Ejecución de servicios mediante la API de OSCAR

Para integrar Argo Workflows con los servicios disponibles en OSCAR, es necesario realizar algunas configuraciones específicas, dependiendo del tipo de servicio a ejecutar. En el caso de servicios accesibles a través de la API REST de OSCAR, la ejecución requiere incluir un *token* de autorización en las solicitudes HTTPS.

Gracias a que Argo Workflows se ejecuta sobre Kubernetes, se puede aprovechar el sistema nativo de secretos de Kubernetes para almacenar y gestionar estos *tokens* de forma segura. Los objetos de tipo *Secret* en Kubernetes permiten almacenar y administrar información confidencial, como contraseñas, *tokens* y llaves de acceso. Ubicar esta información en un *Secret* es más seguro y flexible que ponerlo en la definición de un *pod* o en una imagen [37]. En lugar de incluir el *token* directamente en el cuerpo del *workflow* (lo cual representaría un riesgo de seguridad), se recomienda crear un objeto *Secret* en Kubernetes que contenga el *token* y luego referenciarlo dentro del *workflow* como un volumen montado. En la Figura 15 se puede observar el comando utilizado para crear un secreto que almacena el valor del *token* requerido para la ejecución de un servicio en OSCAR mediante la API, dicho secreto puede ser referenciado desde una plantilla en Argo Workflows, permitiendo ejecutar el servicio referenciando el token sin exponer el valor de este en la plantilla. Este paso es una configuración adicional, útil si se tiene acceso al clúster, sin embargo, no es un paso obligatorio para poder orquestar o ejecutar los servicios. Tomando en cuenta que Argo puede cargar valores a partir de volúmenes montados en el contenedor, se pueden considerar métodos alternativos para almacenar y cargar valores, pero se eligió el método de utilizar secretos Kubernetes por su simpleza y disponibilidad.

```
kubectl create secret -n argo generic cowsay-token --from-
literal=token="61a7899e65a848e1efcfd906e2f7ca66079ed59e24d376a6fab77cf5d948dfcb"
```

Figura 15: Comando utilizado para crear el secreto para la ejecución del servicio Cowsay

Por ejemplo, el *workflow* definido en la Figura 16 ejecuta el servicio Cowsay [44] (previamente creado en OSCAR) mediante una llamada POST a la API. El *token* necesario para la autenticación se monta como un archivo dentro del contenedor, desde donde es leído por el *script* de ejecución. Este enfoque aprovecha la compatibilidad total de Argo Workflows con la sintaxis de volúmenes y secretos definida en las especificaciones de Kubernetes.

```
# cowsay-single-sync.yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  # Prefijo para el nombre del workflow, el sufijo será generado por argo
```

```yaml
  generateName: cowsay-call-
spec:
  # Punto de entrada del workflow
  entrypoint: call-cowsay
  # Los argumentos del workflow cargan el secreto con el token del servicio
  arguments:
    parameters:
      - name: service-token
        # Utiliza un secreto de Kubernetes para obtener el token del servicio
        valueFrom:
          secretKeyRef:
            name: cowsay-token
            key: token
  templates:
    # Definición de la plantilla principal
    - name: call-cowsay
      inputs:
        parameters:
          - name: service-token
      container:
        # Imagen de contenedor que se utilizará
        image: curlimages/curl:latest
        # Comando principal del contenedor: shell
        command: [sh, -c]
        # Argumento del shell: un comando curl para hacer una petición POST
        args:
          - |
            curl -X POST http://oscar.oscar.svc.cluster.local/run/cowsay \
                -H "Authorization: Bearer {{inputs.parameters.service-token}}" \
                -H "Content-Type: application/json" \
                -d '{"message": "Hi there"}'
```

Figura 16: Definición de un workflow que obtiene el token a partir de un secreto para llamar un servicio

Utilizar secretos en lugar de insertar el *token* directamente en el código del *workflow* es la práctica recomendada por motivos de seguridad y mantenimiento. Esto evita exponer información sensible en los manifiestos de Argo Workflows (que suelen almacenarse en sistemas de control de versiones) y permite cambiar o revocar el *token* sin necesidad de modificar el *workflow*. Además, los secretos de Kubernetes pueden ser configurados con restricciones de acceso, evitando su exposición a *pods* o usuarios no autorizados [37].

3.3.2 Ejecución de servicios mediante la carga de un archivo

En OSCAR es posible configurar servicios que se activan automáticamente al detectar la carga de archivos a una carpeta específica dentro de un *bucket* de MinIO (eventos). Una vez desencadenado, el servicio procede a realizar algún tipo de procesamiento sobre el archivo y, en la mayoría de los casos, genera una salida que se almacena en una carpeta designada del mismo u otro *bucket*. No obstante, este comportamiento puede variar dependiendo del tipo de servicio configurado.

Al diseñar un *workflow* que utilice este tipo de servicios basados en eventos, es fundamental tener en cuenta que su ejecución es de naturaleza asíncrona. Es decir, una vez que el archivo es cargado al *bucket* y el servicio se inicia, no se puede asumir que la tarea se ha completado de inmediato. Por lo tanto, se requiere un mecanismo que permita detectar o confirmar la finalización del procesamiento antes de continuar con los pasos siguientes del *workflow*.

Para el caso de ejecutar funciones mediante la carga de un archivo, una de las opciones posibles es utilizar la técnica de *polling* como mecanismo para controlar estos pasos asíncronos. El *polling* consiste en realizar consultas periódicas a una fuente de datos (en este caso, al *bucket* de MinIO) para verificar si se ha producido un cambio esperado, como la aparición de un archivo de salida. Esta estrategia permite que el *workflow* espere activamente a que el resultado del procesamiento esté disponible, sin necesidad de depender de notificaciones externas.

Así, se creó un *workflow* que ejemplifica el proceso de activación y sincronización con un servicio desplegado en OSCAR. Dicho flujo está disponible en el repositorio con el nombre de *grayify-dogs.yaml* [48]. Este flujo está diseñado para copiar un conjunto de archivos desde un *bucket* en MinIO hacia otro *bucket* que actúa como punto de entrada para el servicio *grayify* [55], el cual ha sido previamente desplegado en OSCAR y está basado en *ImageMagick* [56], una herramienta de código abierto para procesamiento de imágenes. El servicio *grayify* transforma imágenes de color a escala de grises de manera automática cuando detecta archivos en su carpeta de entrada. Cabe destacar que, en este caso, el servicio utilizado cumple un propósito meramente ilustrativo; lo verdaderamente relevante es la lógica del patrón de ejecución, basada en activación mediante eventos y la coordinación entre servicios, independientemente de la función específica que desempeñe el servicio procesado.

Este *workflow* está estructurado en dos pasos secuenciales, con el objetivo de mantener una arquitectura modular y controlada (Figura 17).

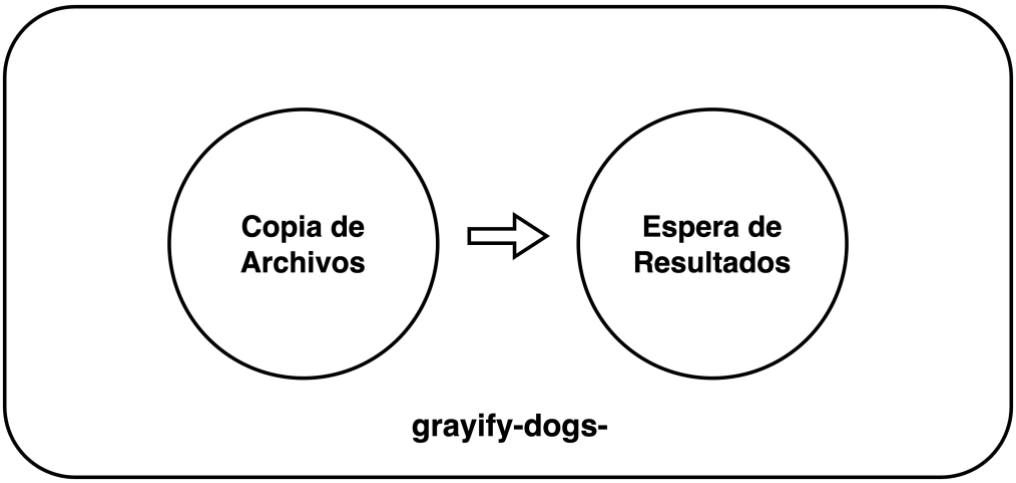


Figura 17: Workflow que convierte un grupo de imágenes a escala de grises

1. **Copia de archivos (copy-to-grayify):** utiliza el cliente de MinIO para transferir imágenes desde un *bucket* de entrada (*test-workflow/in/test-dogs*) hacia la carpeta de entrada del servicio (*grayify/in/*), lo cual dispara automáticamente el procesamiento por parte de grayify.

2. **Espera de resultados (wait-for-grayify):** tras la copia, el *workflow* inicia un proceso de espera activa que supervisa la aparición de los archivos procesados en la carpeta de salida (*grayify/out/*). Este paso se apoya en un *script* personalizado que verifica periódicamente si los resultados están disponibles, con un tiempo máximo de espera definido (*timeout*).

Para la declaración de este flujo se utilizaron WorkflowTemplates, un concepto en Argo Workflows para definir pasos o tareas de un *workflow* y reutilizarlos en diferentes partes o en distintos *workflows* [57]. En lugar de repetir la misma lógica cada vez, se guarda una plantilla con esa lógica y luego se puede usar simplemente haciendo referencia a ella.

La principal ventaja de usar WorkflowTemplates es que ahorran tiempo y evitan errores. Si hay una tarea que se repite muchas veces, como procesar datos o subir archivos, se puede definir una sola vez y reutilizarla cada vez que se necesite. Además, si en algún momento hay que hacer un cambio, solo se modifica la plantilla y ese cambio se aplica automáticamente en todos los *workflows* que lo usan.

También ayudan a tener un sistema más ordenado, donde las tareas están separadas y son fáciles de entender. Esto es muy útil cuando varias personas trabajan en el mismo proyecto o cuando se quiere mantener el sistema a largo plazo sin que se vuelva difícil de manejar.

La plantilla copy-files-template [49] copia archivos recursivamente de una ruta a otra en MinIO. Esta plantilla se utiliza en el *workflow* mencionado previamente [48] para copiar archivos de la ruta *test-workflow/in/test-dogs* a la ruta

grayify/in, pero al ser definido como una plantilla, se puede reutilizar en cualquier otro *workflow* como un paso para copiar archivos de rutas distintas. En la plantilla wait-for-files-template [50] se encuentra la definición de la plantilla que realiza la espera al resultado del procesamiento de las imágenes mediante *polling*.

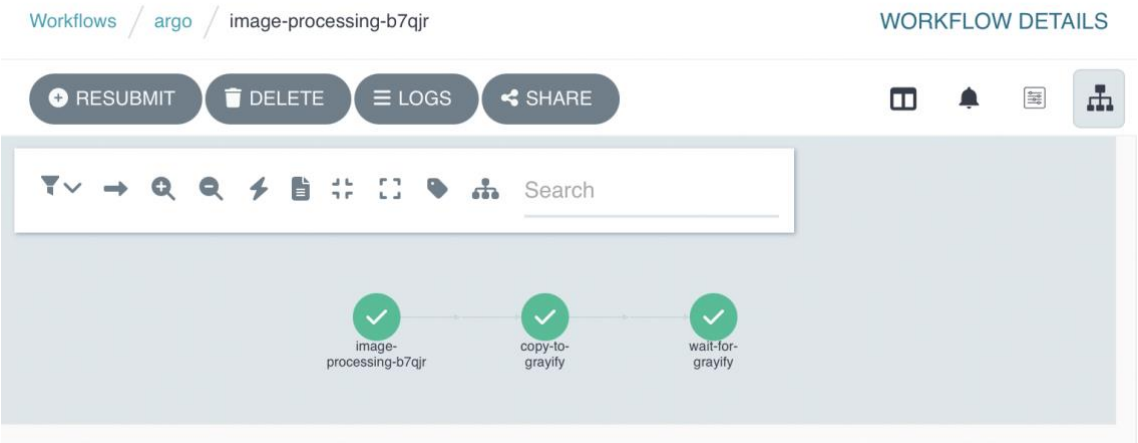


Figura 18: Resultado de la ejecución del workflow image-processing-b7qjr

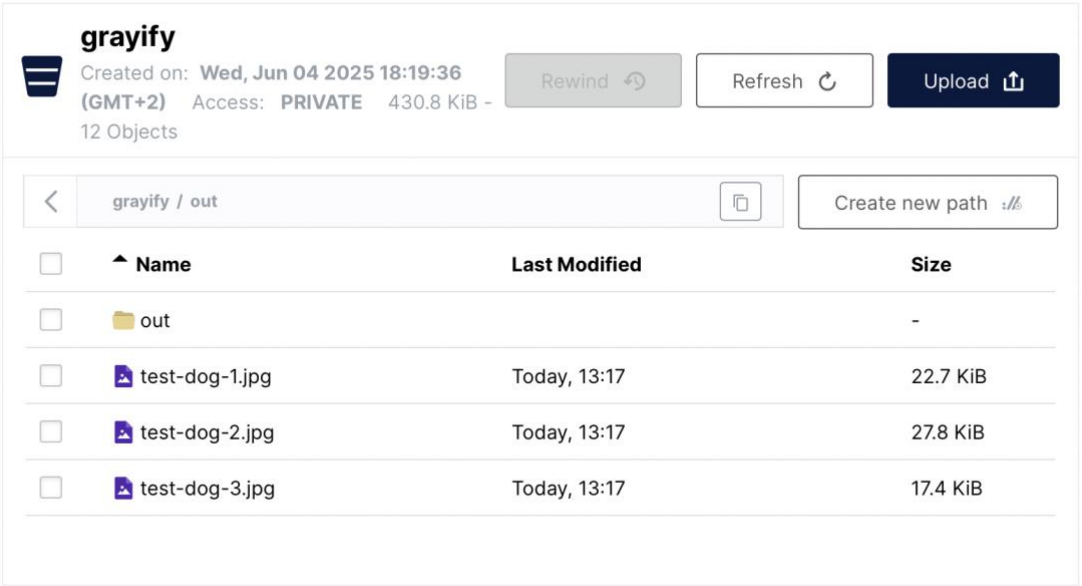


Figura 19: Los archivos resultantes de la ejecución del workflow image-processing-wg9v2

La Figura 18 muestra la información sobre el estado de la ejecución del *workflow*, para el cual se generó el identificador *image-processing-wg9v2*. Podemos observar en dicha figura que el *workflow* consta de dos pasos, el primero es copiar los archivos a la carpeta de entrada y el segundo consiste en esperar los resultados en la carpeta de salida. En la Figura 19 podemos observar que dichos archivos están disponibles en el *bucket grayify/out*, desde donde se pueden descargar para su revisión y futuro uso.

Aprovechando las plantillas previamente creadas, podemos construir otro flujo (Figura 20) que ejecute dos servicios OSCAR, el servicio *grayify* previamente mencionado, y el servicio *plant-classification-theano* [51], servicio que realiza inferencia a partir de la foto de una planta y produce una lista de posibles especies a la que la planta podría pertenecer. El flujo será el mismo que en el *workflow* anterior, repitiendo los pasos entre los servicios *grayify* y plant-*classification*-theano. El manifiesto del *workflow* se puede encontrar en el archivo classify-*plants*.yaml dentro del repositorio de *workflows* [52], en la Figura 21 se puede observar el registro de ejecución del *workflow*.

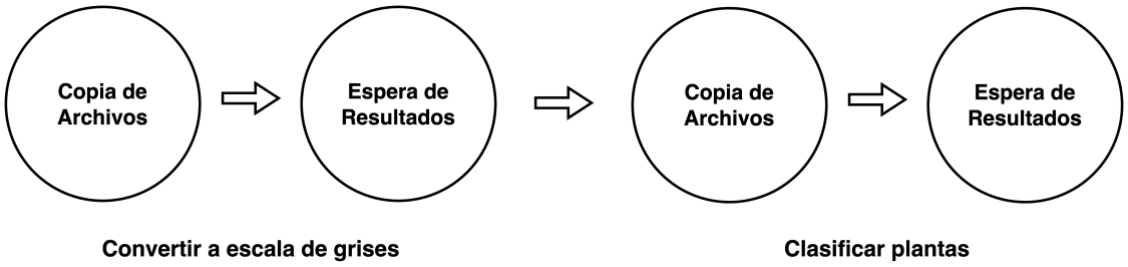


Figura 20: Workflow que convierte imágenes a escala de grises para posteriormente clasificar esas imágenes

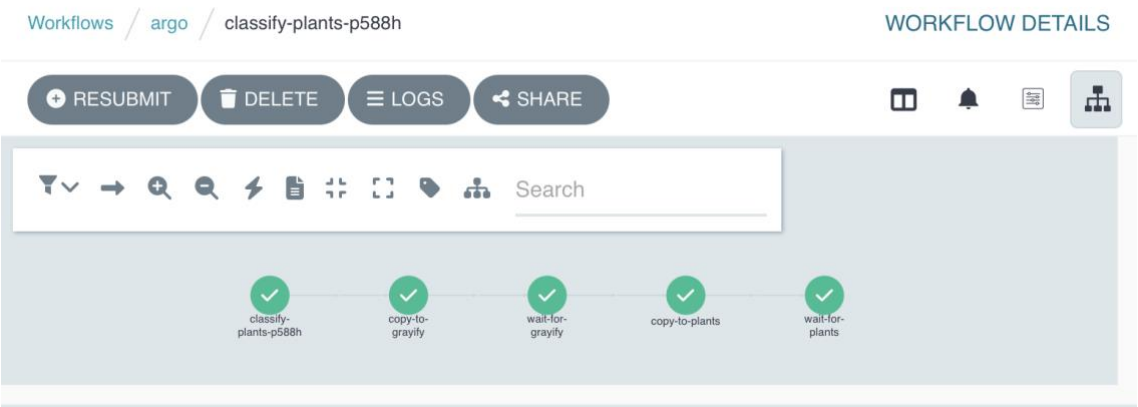


Figura 21: Ejecución del workflow classify-plants

3.3.3 Patrones de ejecución asíncrona

Este apartado aborda los patrones de ejecución asíncrona en la integración entre Argo Workflows y OSCAR, los cuales resultan especialmente útiles cuando se trabaja con servicios que no ofrecen una respuesta inmediata o cuya ejecución depende de eventos externos. Estos patrones permiten suspender temporalmente la ejecución de un *workflow* hasta que se cumpla una condición específica, como la disponibilidad de un archivo en un *bucket* o la recepción de una notificación por parte del servicio. Este enfoque amplía significativamente las capacidades del sistema, permitiendo diseñar *workflows* más robustos y adaptables a escenarios de procesamiento largo o distribuido. A lo largo de esta sección se explicarán los mecanismos empleados para implementar estos patrones, junto con ejemplos concretos y fragmentos de configuración que ilustran su funcionamiento.

Polling

Se distingue por su simplicidad de implementación, ya que permite orquestar los *workflows* sin necesidad de configurar componentes adicionales como notificaciones, sistemas de mensajería o herramientas especializadas como Argo Events [46]. Esta simplicidad reduce de forma considerable la complejidad operativa del entorno y permite concentrarse en la lógica del procesamiento de datos.

Además, este enfoque favorece la portabilidad y replicabilidad del sistema: cualquier usuario con acceso a un clúster compatible puede desplegar y ejecutar los *workflows* sin requerir infraestructura adicional, lo que lo convierte en una solución práctica y accesible para entornos académicos o de investigación.

Suspend/Resume

Argo Workflows cuenta con un mecanismo para los casos en los que se ejecuta un servicio externo asíncrono (por ejemplo, un servicio OSCAR asíncrono o una función lambda en AWS). Este mecanismo consiste en combinar un paso de activación que inicia el trabajo, con un paso de suspensión, que se reanuda mediante una llamada API a Argo Workflows cuando se completa el trabajo externo. El patrón implica dos pasos: el primer paso es un paso de ejecución corta que activa un trabajo de ejecución larga fuera de Argo Workflows (por ejemplo, una petición HTTP), y el segundo paso es un paso de suspensión que suspende la ejecución del *workflow* y, en última instancia, se reanuda o se detiene (es decir, falla) a través de una llamada a la API de Argo Workflows cuando el servicio externo tiene éxito o falla [65].

Este patrón resulta especialmente útil en escenarios donde se invocan servicios externos que pueden tardar un tiempo impredecible en completarse, como tareas de procesamiento intensivo, análisis en lotes o inferencias sobre modelos de aprendizaje automático. A diferencia de enfoques como el *polling*, esta técnica evita la sobrecarga innecesaria de recursos y reduce la complejidad de los *workflows*, ya que el sistema permanece suspendido hasta que recibe una notificación explícita de reanudación. Esta arquitectura reactiva permite diseñar flujos más eficientes y cercanos a un modelo orientado a eventos. También permite implementar mecanismos más confiables de manejo de errores, ya que el *polling* no permite detectar errores directamente en la ejecución (solo permite definir *timeouts*), sin embargo, las notificaciones mediante la API de Argo Workflows permiten detener el *workflow* o un paso específico indicando que sucedió un error en la ejecución.

En la Figura 22 se muestra la estructura de la petición PUT que se hace para reanudar el nodo de un *workflow* mediante la API de Argo Workflows, en el cuerpo se puede observar un campo llamado *nodeFieldSelector*, dicho campo es el que permite identificar el nodo a reanudar, de lo contrario se renaudará todo el flujo [47]. Poder reanudar nodos específicos nos ayuda a tener un mayor control de la orquestación del flujo, permitiendo reanudar nodos paralelos de una

misma tarea. En la Figura 23 se detalla la solicitud para detener un nodo en suspensión, lo que nos permite tener detección de errores desde el servicio ejecutado.

```
curl --request PUT \
 --url https://<URL>/api/v1/workflows/<NAMESPACE>/<WORKFLOWNAME>/resume
 --header 'content-type: application/json' \
 --header "Authorization: $ARGO_TOKEN" \
 --data '{
    "namespace": "<NAMESPACE>",
    "name": "<WORKFLOWNAME>",
    "nodeFieldSelector": "inputs.parameters.uuid.value=<UUID>"
  }'
```

Figura 22: Solicitud HTTPS que resume un nodo del workflow

```
curl --request PUT \
 --url https://<URL>/api/v1/workflows/<NAMESPACE>/<WORKFLOWNAME>/stop
 --header 'content-type: application/json' \
 --header "Authorization: $ARGO_TOKEN" \
 --data '{
    "namespace": "<NAMESPACE>",
    "name": "<WORKFLOWNAME>",
    "nodeFieldSelector": "inputs.parameters.uuid.value=<UUID>",
    "message": "<FAILURE-MESSAGE>"
  }'
```

Figura 23: Solicitud HTTPS que detiene un nodo del workflow

Soporte para eventos

Argo Workflows, por sí mismo, no tiene soporte nativo para reaccionar a eventos externos durante la ejecución de un *workflow*. Su diseño se centra en la orquestación de pasos secuenciales o paralelos, pero no incluye mecanismos integrados para reaccionar a eventos (más allá del *suspend/resume* antes mencionado).

Argo Events, su complemento, sí permite iniciar *workflows* en respuesta a eventos (ej., un nuevo archivo en MinIO o un mensaje en Kafka), pero no extiende esta capacidad a pasos intermedios dentro de un *workflow* ya en ejecución. Esto significa que, para esperar de manera asíncrona durante varios pasos dentro de un *workflow*, la opción más simple y compatible es usar *suspend/resume* o

polling, ya que no requiere infraestructura adicional y se integra directamente con las capacidades de reintento y manejo de errores de Argo Workflows.

Sin embargo, sí es posible ejecutar *workflows* desde otros *workflows* sin necesidad de Argo Events, mediante el uso de la API de Argo Workflows, utilizando el *endpoint /api/v1/workflows/argo/submit* como lo especifica la documentación [53]. Sin embargo, no es una solución tan robusta y con respuesta a tantos tipos de eventos como lo es Argo Events.

Método	**Ventajas**	**Desventajas**
Polling	Sin dependencias externas, fácil depuración y reintentos automáticos	Mala escalabilidad y consumo de recursos en esperas largas
Suspend/ Resume	Notificaciones instantáneas, eficiente en recursos	Requiere configuración adicional para los servicios externos que llaman a la API
Eventos	Escalable, resiliente, resistente a reinicios del clúster	Requiere Argo Events y otros servicios de persistencia como un servidor de colas o mensajería

Figura 24: Comparación entre los distintos métodos de espera o respuesta ante la finalización del servicio

La Figura 24 presenta una comparación entre tres enfoques distintos para gestionar la espera o la respuesta ante la finalización de un servicio en workflows orquestados con Argo Workflows. Cada método se analiza en función de sus principales ventajas y desventajas.

En conclusión, la comparación entre los distintos métodos de espera y notificación ante la finalización de servicios en *workflows* pone de manifiesto que no existe una solución única aplicable a todos los contextos. Cada método ofrece ventajas claras en función de las prioridades del sistema. Mientras que el *polling* destaca por su sencillez y facilidad de implementación, su ineficiencia en escenarios con largas esperas lo limita a casos muy puntuales. Por otro lado, el patrón *suspend/resume* ofrece una solución eficiente y de bajo consumo, aunque exige cierta complejidad en la integración de los servicios externos que deben interactuar con la API de Argo. Finalmente, el enfoque basado en eventos se perfila como el más robusto y escalable, ideal para sistemas distribuidos que requieren alta resiliencia, pero a costa de una mayor complejidad en la infraestructura. La elección entre estas alternativas dependerá, en última instancia, del equilibrio que cada proyecto necesite entre simplicidad, eficiencia y capacidad de escalado.

4. Validación

Este apartado presenta las pruebas realizadas para comprobar el correcto funcionamiento del sistema propuesto, basado en Argo Workflows y orientado a la ejecución de flujos complejos que integran servicios externos a través de OSCAR.

4.1 Objetivo

El objetivo es demostrar que el *workflow* diseñado cumple con los requisitos definidos, tanto a nivel funcional como en aspectos relacionados con la orquestación, el paralelismo, el manejo de eventos externos y el control de errores.

El *workflow* validado corresponde a un pipeline de análisis de sentimiento (Figura 25), que incluye tareas como la recolección de datos desde redes sociales, el procesamiento de archivos CSV, la ejecución de servicios asincrónicos y la generación de resultados estructurados. Durante la validación se pusieron a prueba distintos tipos de espera (*polling*, ejecución síncrona y el patrón *suspend/resume*), así como la ejecución en paralelo de tareas independientes y la gestión de fallos controlados en diferentes puntos del flujo.



Figura 25: Definición simple del workflow de validación

La validación se centró en comprobar que el DAG se ejecuta de forma ordenada, que los pasos se activan y esperan correctamente según el tipo de servicio involucrado, y que el sistema responde adecuadamente en caso de errores.

4.2 Entorno

La ejecución del *workflow* se realizó en un clúster OSCAR desplegado en la infraestructura Cloud basada en OpenStack del grupo de investigación GRyCAP de la Universitat Politècnica de València, el cual está siendo utilizado para el desarrollo de un TFM. Al clúster le fue instalado Argo Workflows, por lo cual se puede observar que en los manifiestos se utilizó directamente el *hostname* del servicio Kubernetes que identifica el punto de acceso a los servicios de MinIO o la API de OSCAR, debido a que la comunicación sucede directamente dentro del clúster entre los componentes involucrados en la ejecución del flujo. Se utilizó OSCAR en su versión 3.6.1 y Argo Workflows en su versión 3.6.7.

Componente	**vCPU**	**RAM**	**HD**	**Nodes**
Frontend	4	16GB	30HD	1
Working Nodes	4	30GB	20HD	1

Figura 26: Recursos del clúster OSCAR

La Figura 26 presenta un resumen de la configuración del clúster utilizado para la integración y validación del sistema. Este clúster está compuesto por dos componentes principales. El primero es el nodo de *frontend*, que dispone de 4 vCPU, 16 GB de memoria RAM y 30 GB de almacenamiento, y cumple la función de coordinar las operaciones y servir como punto de acceso al clúster. El segundo componente es el nodo de trabajo, que cuenta con 4 vCPU, 30 GB de RAM y 20 GB de almacenamiento, siendo responsable de ejecutar las cargas de trabajo y los servicios desplegados. Ambos nodos están desplegados en un único ejemplar, lo que refleja la naturaleza controlada y de pequeña escala del entorno utilizado para este proyecto. Esta configuración es adecuada para tareas de validación y desarrollo, aunque naturalmente impone ciertas limitaciones de capacidad para cargas altamente concurrentes o de gran escala.

4.3 *Workflow*

El *workflow* desarrollado tiene como objetivo principal ejecutar un análisis de sentimiento a partir de comentarios obtenidos de Reddit, integrando distintas etapas que permiten automatizar el proceso completo desde la recolección de datos hasta la clasificación de los comentarios. Está diseñado como un *pipeline* estructurado en forma de DAG, en el que cada paso cumple una función específica y se conecta con el siguiente de manera secuencial o paralela, según el caso.

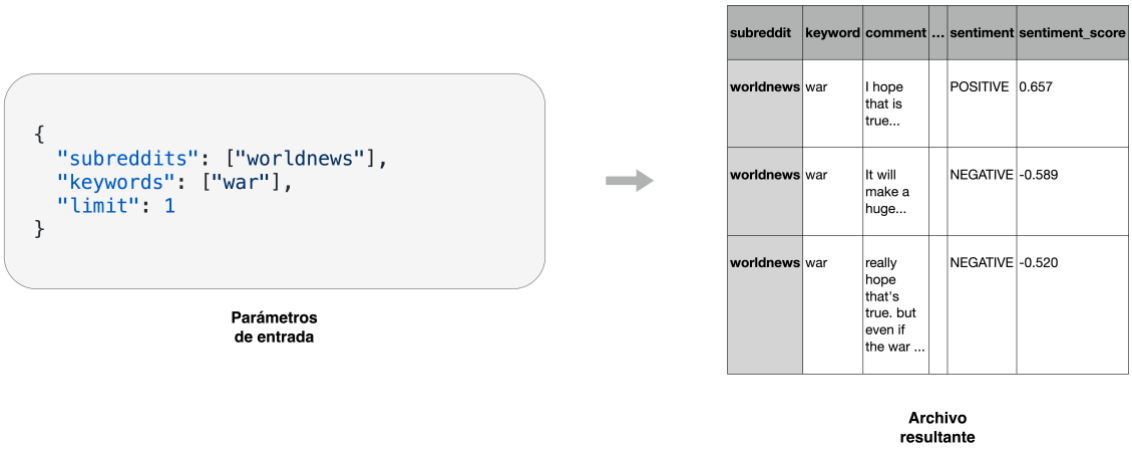


Figura 27: Entrada y sálida del workflow

En la Figura 27 se muestra un ejemplo de la entrada que espera el *workflow* y el archivo de salida que tendrá al finalizar su ejecución. La entrada es un objeto JSON que detalla los parámetros deseados para el análisis de sentimientos. Como salida se obtiene un archivo que contiene el análisis de sentimientos para cada comentario recolectado, a partir de dicho archivo se pueden hacer representaciones gráficas, análisis o modelados adicionales.

4.3.1 Pasos del *workflow*

A continuación, se detallan los pasos del *workflow* representado en la Figura 28, describiendo las tareas realizadas en cada paso y mencionando las herramientas utilizadas en cada uno de ellos.

Recolectar comentarios de Reddit

El primer paso del pipeline es la recolección de datos. Para ello se utiliza PRAW [58], una librería de Python que facilita la interacción con la API de Reddit, siguiendo sus especificaciones oficiales. Esta etapa se encarga de obtener comentarios publicados en los *subreddits* indicados en los parámetros de entrada, realizando búsquedas por palabras clave y filtrando los resultados relevantes. El producto de esta etapa es un archivo CSV (*Comma Separated*

Values), donde se almacenan los comentarios recolectados junto con información adicional.

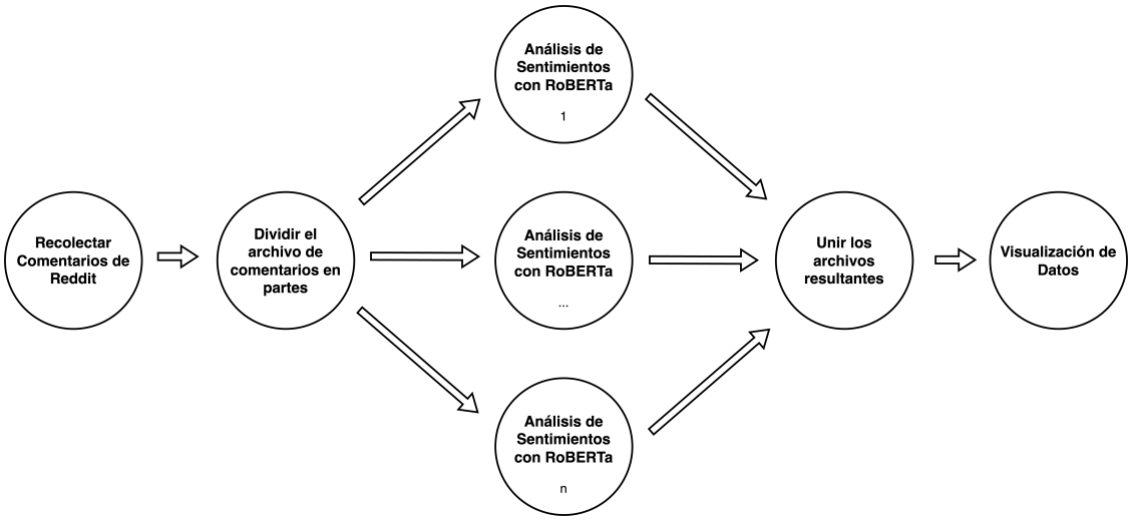


Figura 28: Definición detallada del workflow de validación

Aunque el análisis de sentimientos requiere únicamente el contenido del comentario, se decidió conservar otros campos en el archivo, como el *subreddit*, la palabra clave que lo identificó, el título del post original o la fecha de publicación. Esta información adicional permite realizar análisis más amplios, como observar la evolución temporal del sentimiento respecto a ciertos eventos, o comparar distintas comunidades dentro de Reddit.

Dividir el archivo de comentarios en partes

Al recopilar los comentarios, el tamaño del archivo resultante puede variar mucho según el número de *subreddits*, palabras clave y el límite de publicaciones configurado en la etapa de recolección. Por ejemplo, una búsqueda sencilla que considere un solo *subreddit* y una palabra clave puede generar un archivo con aproximadamente 1,000 registros. En cambio, búsquedas más amplias que incluyan varios *subreddits*, múltiples palabras clave y un límite de hasta 10 publicaciones por *subreddit* pueden generar archivos de hasta 15,000 registros.

Dado que el análisis de sentimientos se realiza de forma individual para cada comentario, el modelo debe ejecutar una inferencia por cada uno de ellos, es decir, se realizan n inferencias donde n es el número total de comentarios. Por esta razón, es necesario considerar estrategias que permitan distribuir la carga de procesamiento. En este flujo, parte del preprocesamiento consiste en dividir el archivo original en varios fragmentos más pequeños, lo que permite distribuir el análisis entre distintas instancias del servicio de inferencia. Esto facilita el paralelismo y mejora el rendimiento general del sistema, y se logra a través de un patrón llamado *fan-out/fan-in,* ejemplificado en la Figura 29.

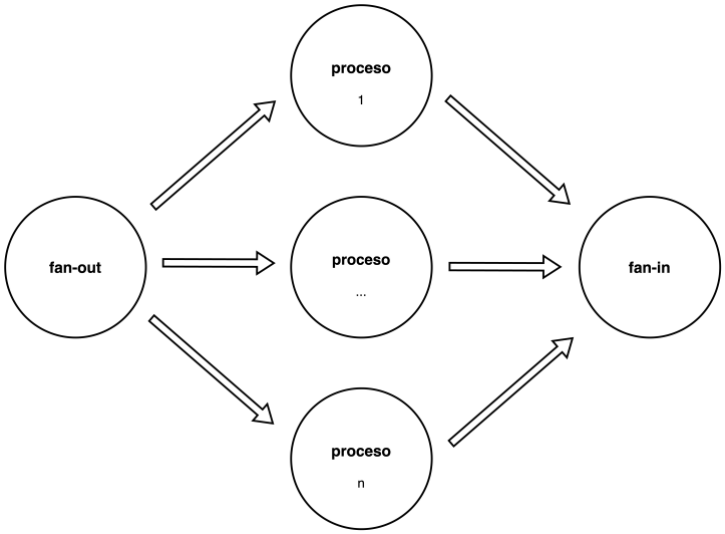


Figura 29: Patrón fan-out/fan-in

El patrón *fan-out/fan-in* es un enfoque que permite dividir un proceso en múltiples tareas que se ejecutan simultáneamente y, posteriormente, unir sus resultados al finalizar. En la etapa *fan-out*, se lanzan varias tareas en paralelo para trabajar de forma concurrente. Luego, en la etapa *fan-in*, se espera a que todas las tareas finalicen y se integran sus salidas. Este enfoque mejora el rendimiento al reducir los tiempos de ejecución mediante la paralelización, además de facilitar la agregación de resultados. Es una técnica eficiente y escalable que favorece un uso óptimo de los recursos computacionales [54].

Análisis de sentimientos con RoBERTa

Para realizar el análisis de sentimientos, se utiliza el modelo RoBERTa (Robustly optimized BERT approach). RoBERTa es un modelo de lenguaje desarrollado por Facebook AI en 2019. Es una versión mejorada de BERT, uno de los modelos más influyentes en procesamiento de lenguaje natural (NLP) [59]. Ambos se basan en la arquitectura de transformadores, pero RoBERTa introduce varias mejoras clave. Para su implementación, se emplean principalmente las librerías Transformers [60] de Hugging Face y PyTorch [61], debido a su integración eficiente y compatibilidad con modelos basados en arquitecturas de tipo transformador.

Transformers

La librería Transformers proporciona una interfaz de alto nivel para el uso de modelos de lenguaje pre-entrenados, como RoBERTa. Esta librería permite:

- Cargar fácilmente el modelo RoBERTa y su *tokenizador* correspondiente desde un repositorio pre-entrenado.
- Realizar el preprocesamiento del texto, incluyendo la *tokenización* y la conversión a tensores adecuados para el modelo.
- Ejecutar la inferencia del modelo.

PyTorch

La librería PyTorch actúa como motor de ejecución para el modelo. Es responsable de:

- Representar los datos como tensores, permitiendo su procesamiento eficiente tanto en CPU como en GPU.
- Ejecutar el modelo RoBERTa durante la fase de inferencia o entrenamiento.
- Gestionar el flujo de datos a través de las capas del modelo y controlar el uso de recursos del CPU.

Unir los archivos resultantes
Al finalizar la ejecución de todos los procesos de análisis de sentimiento, es necesario reunir los archivos resultantes de esos procesos, por eso existe un paso al finalizar que reúne los archivos CSV resultantes en un archivo unificado, lo cual permite realizar otras tareas de análisis sobre los resultados del análisis de sentimiento.

Visualización de datos
Una vez completado el análisis de sentimientos sobre los comentarios recolectados, el archivo resultante (en formato CSV) es utilizado para actualizar un repositorio de datos público en GitHub, específicamente en *o4tmeal/sentiment-analysis-dashboard* [62]. Este repositorio contiene tanto el archivo de resultados como un *script* en Python que implementa un *dashboard* interactivo utilizando la librería Streamlit.

El *dashboard* permite visualizar los resultados del análisis de manera clara y dinámica, facilitando la exploración de tendencias, comparativas entre *subreddits*, evolución temporal del sentimiento, entre otros aspectos relevantes. La plataforma Streamlit Cloud, que permite desplegar aplicaciones creadas con Streamlit directamente desde repositorios de GitHub, se encarga de alojar el *dashboard* de forma continua [63].

Gracias a esta integración, cada vez que se actualiza el archivo CSV en el repositorio (por ejemplo, al ejecutar nuevamente el *workflow* con nuevos parámetros de búsqueda), los cambios se reflejan automáticamente en la interfaz del *dashboard* sin necesidad de hacer un despliegue o intervención manual. Este enfoque introduce elementos de integración y entrega continuas (CI/CD), al automatizar tanto la actualización del conjunto de datos como la visualización en producción. La integración con GitHub y Streamlit Cloud permite que cualquier ejecución del *workflow* se refleje automáticamente en el *dashboard* publicado, eliminando la necesidad de tareas manuales y garantizando una visualización siempre actualizada.

4.3.2 Diseño y definición del *workflow*
Para validar la propuesta de orquestación de *workflows serverless*, se diseñó e implementó un *workflow* completo que refleja un caso de uso realista y actual: el análisis de sentimiento de comentarios publicados en Reddit.

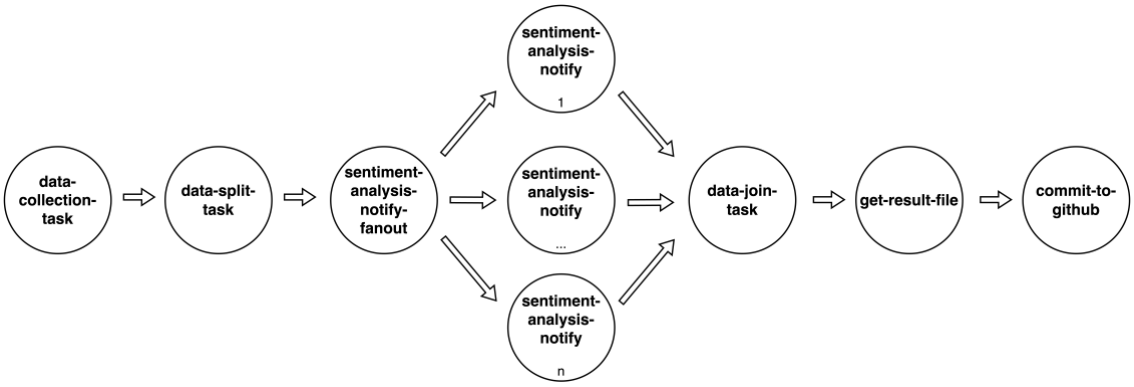


Figura 30: Representación gráfica del DAG definido en el workflow

El *workflow*, disponible como plantilla en el repositorio [64], fue definido como un DAG, aprovechando las capacidades de paralelismo, control de errores y sincronización que ofrece la herramienta. Este apartado detalla el diseño lógico del flujo, los objetivos de cada etapa, los patrones implementados y las decisiones técnicas que sustentan su funcionamiento. En la Figura 30 se puede observar un grafo que representa cada uno de los pasos definidos en el manifiesto del *workflow*.

Se utilizará la siguiente estructura para describir cada paso del flujo, con la intención de proveer el mayor detalle posible sobre la definición de cada paso del flujo y explicar lo que conlleva el diseño y creación de un *workflow* en OSCAR.

- *Script* del paso a ejecutar: Se facilitará la referencia al *script* desarrollado que contiene la lógica a ejecutar como servicio OSCAR.
- Definición FDL del servicio: La definición del servicio OSCAR en formato FDL, necesaria para crear el servicio en el clúster OSCAR y definir sus parámetros de ejecución.
- WorkflowTemplate: Plantilla de definición del paso como manifiesto de Argo Workflows, necesario para la creación de una plantilla reutilizable entre *workflows*.

Los archivos completos estarán disponibles en el repositorio para su consulta [65].

Recolectar comentarios de Reddit

El *script recolectar_datos.py* [66] realiza la recolección de datos mediante el uso de la herramienta PRAW, este *script* se empaqueta en un contenedor Docker para poder crear un servicio en OSCAR que ejecute el proceso de recolección a demanda.

```yaml
functions:
  oscar:
    - oscar-cluster:
        name: reddit-collector
        memory: 1Gi
        cpu: '1.0'
        image: o4tmeal/reddit-collector:latest
        script: script.sh
        log_level: CRITICAL
        environment:
          Variables:
            INPUT_TYPE: json
          # Define los secretos a los que tendrá acceso el servicio
          secrets:
            reddit_client_id: "V1wUWTjRtSvEhxdi95Pw"
            reddit_client_secret: "urV4qsu9KDducEPflF9YbPR420"
            reddit_user_agent: "sentiment_analysis_collector"
        input:
          - storage_provider: minio.default
            path: reddit-collector/input
        output:
          - storage_provider: minio.default
            path: reddit-collector/output
```

Figura 31: Definición FDL del servicio recolector de datos de Reddit

El manifiesto FDL [67] (Figura 31) para crear este servicio lo define como un servicio asíncrono, el cual se activa automáticamente cuando se carga un archivo en el *bucket* de entrada de datos. Esta configuración responde al hecho de que el servicio puede tener una duración variable, dependiendo de la cantidad de comentarios recolectados y procesados. En OSCAR, a partir de la versión 3.6.1 se pueden definir valores secretos a los que el servicio tendrá acceso como variables de entorno. En este manifiesto se proporcionan como ejemplo valores falsos para mantener la privacidad.

La plantilla *collect-data-template* define una etapa del *workflow* encargada de preparar y activar el proceso externo de recolección de datos desde Reddit [68]. Esta plantilla utiliza la imagen *bitnami/minio-client* para interactuar con el almacenamiento MinIO, al que accede mediante credenciales configuradas por secreto.

Durante su ejecución, la plantilla realiza las siguientes acciones:

1. Guarda un archivo de configuración JSON recibido como parámetro (config) en el sistema de archivos.
2. Sube este archivo a MinIO, dentro del *bucket* y ruta especificados por el parámetro input, utilizando como nombre el identificador del *workflow* (workflowName).
3. Espera pasivamente hasta que el proceso externo encargado de recolectar datos cree el archivo CSV de salida correspondiente en la ubicación indicada por el parámetro output.

Esta plantilla permite desacoplar el *workflow* del proceso de *scraping* real, facilitando su integración con servicios externos en OSCAR que serán activados por la aparición del archivo de configuración en MinIO.

Dividir el archivo de comentarios en partes

Para dividir el archivo resultante de recolectar los comentarios, se creó un servicio que descarga el archivo de su ubicación en el *bucket* de MinIO, divide el archivo en diferentes partes con un número similar de comentarios por archivo (configurable al momento de activar el servicio), el *script* puede verse en el repositorio del proyecto [69].

El manifiesto FDL [70] define el servicio como un servicio síncrono que recibe la configuración de activación en formato JSON, se definió como un servicio síncrono al ser una tarea breve y de baja latencia.

Esta plantilla, *split-data-sync-template* [71], define una tarea responsable de dividir un archivo CSV en fragmentos más pequeños, de forma sincronizada, mediante la invocación del servicio desplegado en OSCAR. Utiliza un contenedor personalizado con soporte para las herramientas jq (herramienta de línea de comandos para procesar estructuras JSON) y curl (herramienta de línea de comandos para transferir datos desde o hacia un servidor utilizando diversos protocolos), y se comunica con el *endpoint* del servicio *split-csv-sync*, enviando una configuración generada dinámicamente con las rutas de entrada/salida y las credenciales de MinIO. Como resultado, el servicio devuelve la lista de archivos generados, que se extrae del JSON de respuesta y se guarda como una salida de la plantilla. Este componente es esencial en el *workflow* para habilitar el procesamiento paralelo de grandes volúmenes de datos, permitiendo que los fragmentos resultantes se distribuyan a múltiples instancias de análisis en pasos posteriores.

Análisis de sentimientos con RoBERTa

El *script* de análisis de sentimientos [72] implementa un servicio de análisis de sentimientos basado en RoBERTa, diseñado para ejecutarse como un componente en una arquitectura *serverless* paralela orquestada con Argo Workflows y OSCAR. Su función principal es procesar archivos CSV y generar un

nuevo archivo con el sentimiento asociado a cada comentario (positivo, negativo o neutro), junto con una puntuación de confianza.

A continuación, se describe brevemente su funcionamiento:

- Carga de modelo: Utiliza el modelo pre-entrenado *cardiffnlp/twitter-roberta-base-sentiment* para clasificación de sentimientos, cargando también su *tokenizador*.
- Procesamiento de datos: Lee el archivo CSV de entrada especificado por el usuario (vía argumento), localiza la columna de texto a analizar, y aplica el modelo para inferir el sentimiento de cada fila.
- Inferencia y resultados: Para cada texto, calcula las probabilidades de sentimiento y asigna tanto una etiqueta (NEGATIVE, NEUTRAL, POSITIVE) como una puntuación numérica asociada.
- Salida: Guarda el DataFrame resultante en un archivo CSV en la ruta de salida indicada.
- Notificación: Si se proporcionan los datos necesarios para la notificación por HTTP, se notifica tanto si sucede un error o si la ejecución es exitosa.

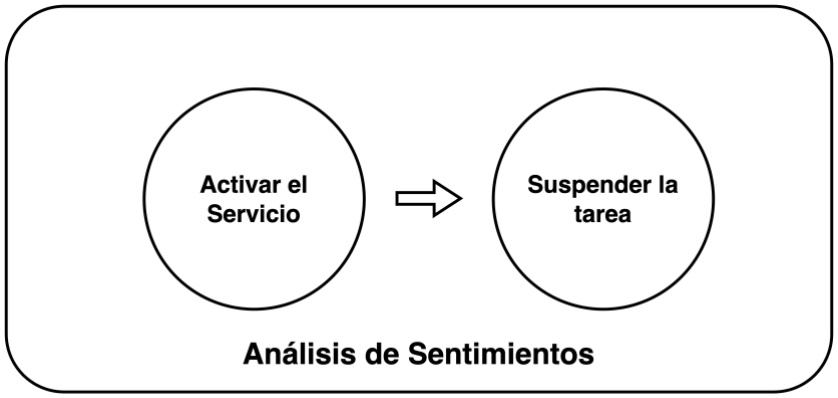


Figura 32: La tarea que orquesta el análisis de sentimientos

El servicio [73] se define como un servicio asíncrono activado mediante la carga de un archivo CSV al *bucket* de entrada. La idea es poder ejecutar de manera asíncrona este servicio de forma paralela para repartir la carga de trabajo, y ser notificado mediante una llamada HTTP a la API de Argo Workflows cuando el servicio finalice su ejecución.

El WorkflowTemplate *sentiment-analysis-notify-template* [74] está diseñado para activar el servicio de análisis de sentimientos de manera asíncrona y esperar su finalización. El flujo consta de dos pasos principales (Figura 32): primero, el paso *activate* ejecuta una plantilla llamada *activate-service*, que genera un archivo de configuración JSON y lo sube a un *bucket* de MinIO. Este archivo contiene la información necesaria para que un servicio externo inicie el análisis del archivo especificado, incluyendo rutas de entrada y salida, credenciales de

MinIO, y URLs de notificación para reanudar o detener el *workflow* dependiendo del resultado. Una vez activado el servicio, el segundo paso *wait* entra en estado de suspensión (*suspend*) hasta recibir una señal de reanudación o terminación desde el exterior, facilitando un patrón de integración tipo "resumir al completar". Este diseño permite desacoplar el análisis intensivo de sentimientos del control del *workflow*, manteniendo el pipeline reactivo y robusto frente a servicios de larga duración. Este paso utiliza la propiedad *withValues*.

Unir los archivos resultantes

El *script* [75] recibe un archivo de configuración JSON que especifica una lista de archivos CSV alojados en MinIO, los descarga, los une en un único archivo consolidado y lo sube nuevamente a MinIO en la ruta de salida indicada. Este servicio se ejecuta automáticamente al completarse tareas paralelas anteriores del *workflow*, permitiendo consolidar resultados distribuidos en una sola salida. Este servicio también se define de forma síncrona [76], dado a que es una tarea breve y no requiere de esperas prolongadas.

La plantilla *join-data-sync-template* [77] facilita la unión de múltiples archivos CSV en un único archivo consolidado, delegando esta tarea a un servicio en OSCAR. A partir de los parámetros de entrada (lista de archivos, ruta de salida, nombre del archivo resultante y *token* de autorización), el contenedor genera dinámicamente un archivo de configuración en formato JSON utilizando jq. Este archivo es enviado mediante una solicitud POST al servicio *join-csv-sync*, que se encarga del procesamiento. Una vez recibida la respuesta, la plantilla extrae la ruta del archivo de salida y la guarda como parámetro de salida para pasos posteriores. Esta encapsulación permite desacoplar el proceso de unión de datos del *workflow* principal, favoreciendo la reutilización, modularidad y separación de responsabilidades.

Visualización de datos

Para actualizar el *dashboard* de visualización, esta etapa se compone de dos pasos (Figura 33). El primero consiste en descargar el archivo generado en el paso anterior, que consolida los resultados de las ejecuciones concurrentes en un único archivo. Este paso fue implementado directamente como un WorkflowTemplate [78], sin necesidad de crear un servicio adicional en OSCAR. Una de las ventajas de que Argo Workflows sea nativo de Kubernetes es la posibilidad de definir y ejecutar tareas directamente, sin requerir servicios externos. Dado que se trata de una operación sencilla, se optó por gestionarla íntegramente desde Argo Workflows.

El segundo paso consiste en subir el archivo consolidado al repositorio de GitHub vinculado al *dashboard* alojado en Streamlit Cloud. Al igual que el paso anterior, este también se definió como un WorkflowTemplate [79], ejecutando la lógica directamente desde Argo Workflows por su simplicidad y eficiencia, sin necesidad de depender de servicios adicionales.

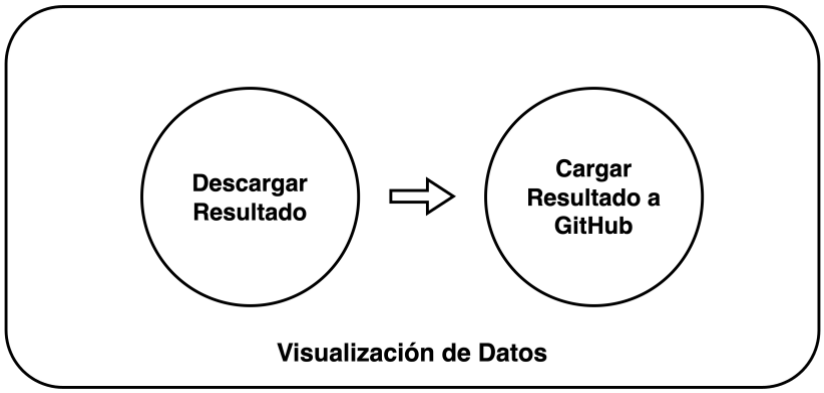


Figura 33: Paso de visualización de datos

4.4 Ejecución

La validación del sistema desarrollado se centró en comprobar el correcto funcionamiento de la integración entre Argo Workflows y OSCAR dentro de un entorno real de ejecución. Esta validación tuvo como objetivo principal asegurar que los distintos componentes se comunican de manera fluida y que el *workflow* completo puede orquestarse de forma declarativa desde Argo Workflows, ejecutando servicios definidos y desplegados en OSCAR.

4.4.1 Ejecución correcta del flujo

Durante las pruebas, se ejecutó múltiples veces el *workflow* con distintos parámetros. El *workflow* ejecutó los pasos asíncronos, llamadas a servicios expuestos por OSCAR mediante API o eventos en MinIO, procesamiento de archivos, espera por resultados, y consolidación final. Se comprobó que Argo Workflows puede suspenderse adecuadamente mientras un servicio OSCAR realiza su tarea, reanudarse posteriormente tras una notificación o condición de sincronización, y continuar con pasos posteriores, lo que valida la capacidad del sistema para coordinar procesos distribuidos de forma confiable.

Además, se evaluó que el sistema puede reutilizar los *workflows* definidos para diferentes parámetros de entrada (Figura 27), sin requerir cambios en el código ni en la lógica del flujo, lo que confirma su flexibilidad. La integración permite encapsular servicios como contenedores, delegar su ejecución a OSCAR, y utilizar Argo Workflows para organizar su encadenamiento, control de errores, patrones de *fan-out/fan-in* y publicación de resultados. Esta sinergia entre ambos sistemas demuestra que es posible construir flujos complejos, mantenibles y portables sin necesidad de herramientas externas adicionales.

En la Figura 34 se puede observar una ejecución exitosa para el flujo *sentiment-analysis-workflow-2q5lt*, el cual genera 11 pasos concurrentes para el análisis de sentimientos, produce un archivo resultante y carga el resultado a GitHub.

En la Figura 35 se muestran los archivos generados por el paso data-split-task para el *workflow sentiment-analysis-workflow-2q5lt*, estos archivos pueden servir para inspeccionar paso a paso los datos producidos por el flujo en el caso

de querer depurar la ejecución. También se podría agregar un paso de salida al flujo que limpiara automáticamente todos los archivos generados por el flujo que no se deseen conservar mediante el uso de *exitHandlers* [80]. En la Figura 36 podemos ver que el archivo se actualizó correctamente en el repositorio, provocando la actualización del *dashboard* interactivo, visible en la Figura 37.

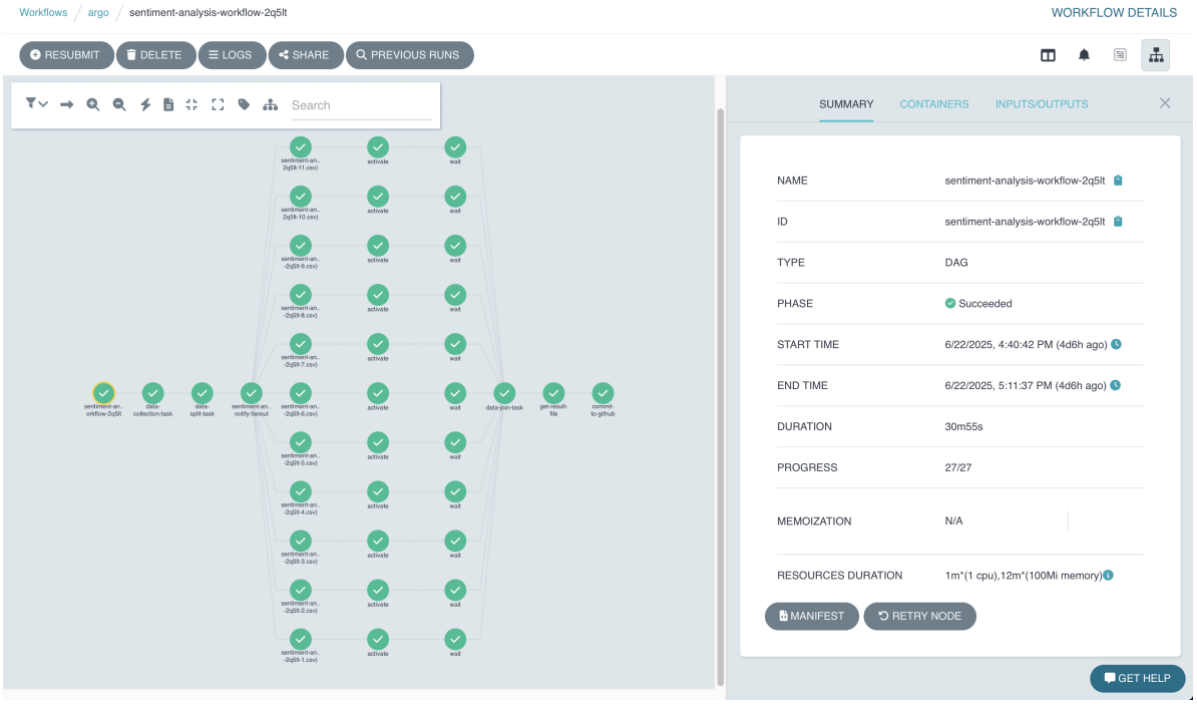


Figura 34: Ejecución del flujo de análisis de sentimientos sentiment-analysis-workflow-2q5lt

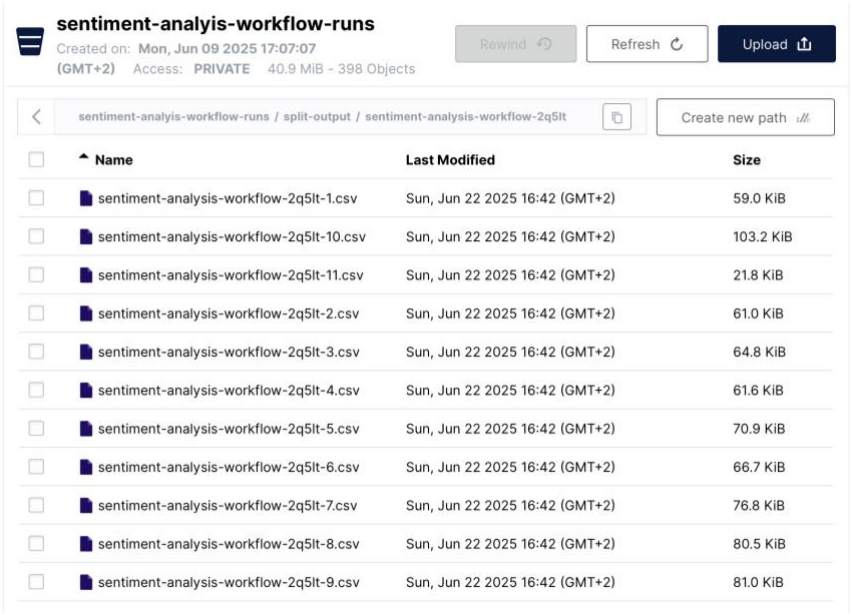


Figura 35: Archivos generados para el paso data-split-task del workflow sentiment-analysis-workflow-2q5lt

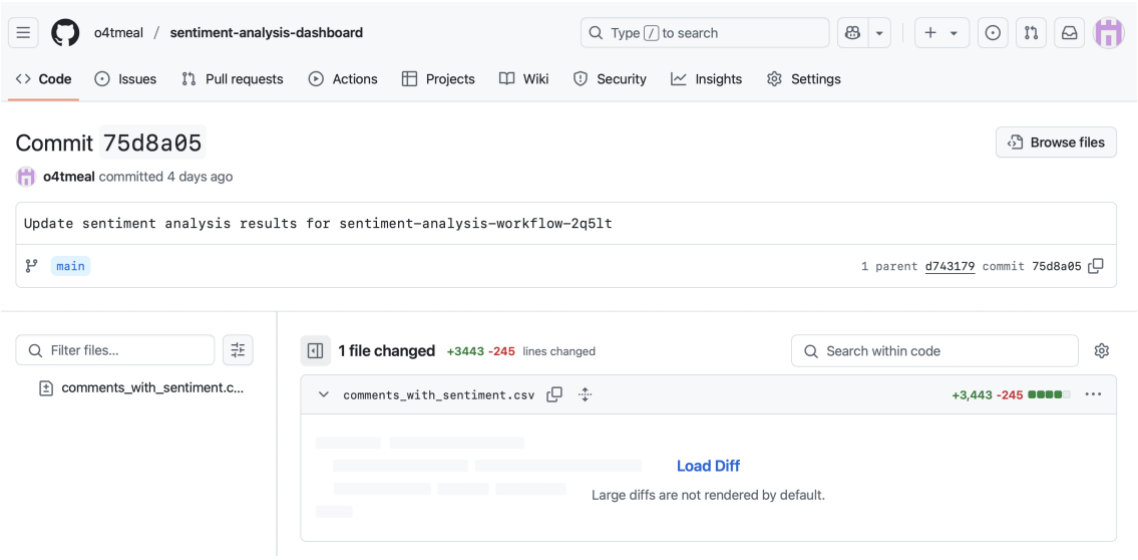


Figura 36: El archivo resultante del workflow sentiment-analysis-workflow-2q5lt se actualiza en el repositorio

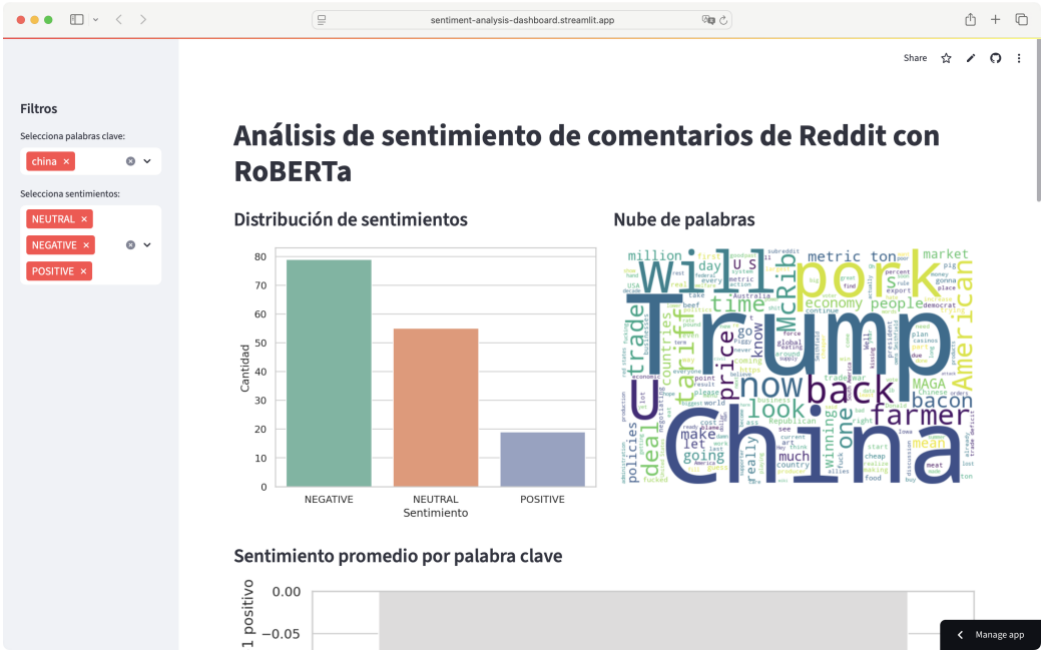


Figura 37: Dashboard interactivo de visualización de datos

4.4.2 Desempeño

En el contexto de este trabajo, la validación de la orquestación del flujo de procesamiento de datos se centró en demostrar la capacidad de automatizar y coordinar múltiples etapas de análisis mediante Argo Workflows y su integración con servicios OSCAR. Si bien uno de los aspectos que naturalmente surge en este tipo de sistemas es la escalabilidad, es importante destacar que, en este caso, la evaluación se llevó a cabo sobre un clúster de recursos limitados, con una configuración que, aunque adecuada para fines de prueba y desarrollo, impone ciertas restricciones al evaluar el comportamiento a gran escala.

Particularmente, el clúster utilizado para este proyecto dispone de una capacidad de cómputo limitada, con solo cuatro núcleos de CPU en total, distribuidos entre un nodo de control y un nodo de trabajo. Debido a que el nodo maestro tiene aplicada una restricción (*taint*) que impide la ejecución de cargas de trabajo ordinarias, todo el procesamiento recae en el nodo de trabajo, que comparte sus recursos entre los servicios desplegados en OSCAR, Argo Workflows y las tareas del *workflow*. Durante la ejecución de pruebas, se observó que, aunque el uso total de CPU indicaba disponibilidad aparente para nuevas tareas, Kubernetes no fue capaz de programar nuevas instancias debido a la segmentación de los recursos y a la imposibilidad de satisfacer completamente las solicitudes de CPU por *pod*. Esta situación refleja un límite práctico en la escalabilidad del sistema bajo las condiciones actuales.

Aunque el *workflow* fue diseñado para admitir una ejecución concurrente de tareas, el rendimiento observado estuvo condicionado por estos cuellos de botella, lo que derivó en tiempos de ejecución crecientes a medida que aumentaba la cantidad de datos procesados. No obstante, es importante destacar que el propósito de esta validación no fue realizar un estudio de rendimiento bajo condiciones óptimas, sino evaluar la capacidad del sistema para coordinar múltiples tareas en un entorno distribuido y *serverless*, y confirmar la viabilidad de su integración.

ID	Número de comentarios	Número de nodos concurrentes	Recursos	Tiempo de ejecución (en minutos)
sentiment-analysis-workflow-2q5lt	2,052	11	6GB RAM 1 vCPU	30
sentiment-analysis-workflow-x984c	5,568	26	6GB RAM 1 vCPU	72
sentiment-analysis-workflow-jgm9x	15,585	76	6GB RAM 1 vCPU	202

Figura 38: Tiempos de ejecución del workflow

Desde esta perspectiva, el sistema logró cumplir con sus objetivos. Argo Workflows y OSCAR se integraron correctamente, permitiendo la ejecución coordinada de pasos, el uso de patrones asíncronos como *fan-out/fan-in*, y la consolidación de resultados, todo ello dentro de un clúster autogestionado. Aunque los tiempos de ejecución reflejan las limitaciones impuestas por el

entorno, estos no afectan la validez de la solución propuesta en cuanto a su arquitectura y funcionamiento general.

Los resultados obtenidos, reflejados en la Figura 38, permiten observar un comportamiento escalable, pero *sublineal*, consistente con lo esperado en arquitecturas distribuidas sobre recursos compartidos. A medida que se incrementó el número de tareas concurrentes, el tiempo total de ejecución disminuyó, pero con un beneficio marginal decreciente, atribuible a la sobrecarga de coordinación, la espera por recursos y la competencia por CPU. La eficiencia observada se vio afectada no por fallos del sistema, sino por la disponibilidad limitada de núcleos y la exclusión del nodo de control del *scheduling* de cargas de trabajo. Este análisis refuerza la idea de que, si bien la plataforma puede escalar, su rendimiento final dependerá directamente de la infraestructura subyacente.

4.5 Conclusiones

Las pruebas realizadas durante la fase de validación demostraron la eficacia y viabilidad de la integración entre Argo Workflows y la plataforma OSCAR para la ejecución de workflows en entornos serverless. A través de casos prácticos concretos, como el análisis de sentimientos de comentarios en Reddit y la clasificación de imágenes, se comprobó que el sistema es capaz de gestionar flujos de trabajo complejos de forma automatizada, utilizando tanto ejecuciones síncronas como patrones de ejecución asíncrona mediante *polling*.

Uno de los aspectos más destacados fue la capacidad del sistema para coordinar múltiples servicios y tareas de manera ordenada, garantizando la correcta ejecución de cada etapa del *workflow* y la disponibilidad de los resultados en el almacenamiento del clúster. Además, el uso de plantillas reutilizables permitió simplificar el diseño de *workflows*, favoreciendo la modularidad y la reutilización de componentes en distintos experimentos.

La validación también evidenció que la solución es reproducible y adaptable, ya que los *workflows* definidos pueden replicarse fácilmente en otros entornos que cuenten con una infraestructura similar basada en Kubernetes y OSCAR. Esto abre la puerta a su uso en escenarios más amplios, tanto en investigación como en producción. Asimismo, el sistema demostró un comportamiento estable durante las ejecuciones, con un consumo de recursos adecuado para un entorno de pruebas.

En conclusión, el apartado de validación confirmó que la arquitectura propuesta no solo es funcional, sino que también ofrece ventajas en términos de automatización, reproducibilidad y escalabilidad, cumpliendo con los objetivos planteados al inicio del trabajo. Los resultados obtenidos respaldan la viabilidad del uso de Argo Workflows como herramienta central para la gestión de *workflows* en entornos *serverless* distribuidos como OSCAR.

5. Conclusiones

Las conclusiones de este trabajo evidencian que la gestión de *workflows* en entornos *serverless* representa un desafío técnico relevante, debido a la naturaleza efímera y sin estado de las funciones involucradas. Esta característica limita la comunicación directa entre tareas y obliga a emplear herramientas externas de orquestación que gestionen las dependencias, el control de errores y la coordinación del flujo completo. En este contexto, la selección de una herramienta adecuada se vuelve clave para garantizar el correcto funcionamiento de aplicaciones distribuidas bajo este modelo.

A lo largo del proceso de evaluación se analizaron diversas herramientas populares de orquestación de *workflows*, como Synapse, SonataFlow, AWS Step Functions, Apache Airflow, Temporal, Cadence y Prefect. Aunque muchas de estas soluciones ofrecían características avanzadas, ninguna se ajustaba completamente a los requerimientos específicos del entorno OSCAR, ya fuera por su complejidad de despliegue, falta de integración con Kubernetes, necesidad de procesos residentes o dependencia de infraestructuras comerciales. Estas limitaciones reforzaron la necesidad de optar por una solución ligera, nativa en Kubernetes y de código abierto.

La elección de Argo Workflows respondió a estos criterios, y su implementación dentro del clúster OSCAR demostró ser una solución eficaz y versátil. Gracias a su enfoque declarativo, su arquitectura basada en contenedores y su compatibilidad con los servicios del clúster, Argo Workflows permitió construir *workflows* reproducibles, escalables y fáciles de mantener. Además, su interfaz gráfica, documentación y comunidad activa facilitaron el desarrollo y validación de flujos complejos como el procesamiento de imágenes o el análisis de sentimientos.

En definitiva, este trabajo no solo valida la viabilidad técnica de usar Argo Workflows en un entorno *serverless* como OSCAR, sino que también ofrece una guía práctica y reproducible para otros proyectos que busquen orquestar tareas bajo este paradigma. Se concluye así que Argo Workflows es una opción especialmente adecuada para entornos académicos o de investigación que requieran soluciones eficientes, flexibles y sin dependencia de proveedores comerciales.

En este sentido, los resultados obtenidos en el desarrollo de este trabajo han permitido alcanzar los objetivos propuestos al inicio, que consistían en analizar la viabilidad de utilizar Argo Workflows como herramienta de orquestación en entornos serverless, evaluando su integración con OSCAR y su capacidad para gestionar flujos de trabajo complejos. A través de la selección, instalación, configuración y validación de workflows, se ha comprobado que Argo Workflows

no solo cumple con los requisitos técnicos definidos, sino que también ofrece una solución práctica, reproducible y adaptable a este tipo de entornos. Por tanto, se puede afirmar que los objetivos planteados han sido cumplidos.

5.1 Trabajos futuros

En este trabajo se ha hecho una exploración técnica en la gestión de *workflows* en entornos *serverless* mediante la integración de Argo Workflows con una plataforma externa de ejecución de servicios. No obstante, existen múltiples líneas de trabajo que pueden desarrollarse a partir de los resultados obtenidos. Una dirección prometedora es la incorporación de herramientas complementarias como Argo Events para permitir una orquestación más reactiva, basada en eventos externos. Esta integración permitiría una mayor flexibilidad en la automatización de tareas, así como una respuesta más eficiente a estímulos externos en tiempo real.

Otra posible línea evolutiva consiste en explorar arquitecturas híbridas o distribuidas en múltiples clústeres, en las que Argo Workflows actúe como orquestador centralizado de tareas ejecutadas en diferentes dominios o infraestructuras. Esta capacidad permitiría validar escenarios de federación de recursos o computación en el borde, en línea con las tendencias actuales de descentralización del procesamiento. Asimismo, sería interesante analizar el rendimiento y la escalabilidad del sistema en cargas de trabajo más intensivas, como *pipelines* de entrenamiento de modelos de aprendizaje automático, flujos de datos en *streaming* o integración con fuentes heterogéneas.

Finalmente, resulta relevante avanzar hacia la estandarización de los flujos definidos en Argo Workflows mediante su traducción automática o interoperabilidad con especificaciones como SWS o CWL. Este esfuerzo podría facilitar la portabilidad y la reutilización de *workflows* entre plataformas, promoviendo la creación de un ecosistema más abierto y sostenible. En conjunto, estas líneas de trabajo contribuirían a fortalecer el rol de Argo Workflows y OSCAR como herramientas clave para la investigación y el desarrollo de soluciones *serverless* avanzadas.

6. Referencias

1. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., ... & Slominski, A. (2017). Serverless computing: Current trends and open problems. Research Advances in Cloud Computing. https://link.springer.com/chapter/10.1007/978-981-10-5026-8_1

2. Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. 2022. Mashup: making serverless computing useful for HPC workflows via hybrid execution. In Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '22). Association for Computing Machinery, New York, NY, USA, 46–60. https://doi.org/10.1145/3503221.3508407

3. Argo Workflows. https://argoproj.github.io/workflows/

4. Kubernetes. https://kubernetes.io/

5. AWS Lambda. https://docs.aws.amazon.com/es_es/lambda/latest/dg/welcome.html

6. OSCAR. https://oscar.grycap.net/

7. Common Workflow Language (CWL). https://www.commonwl.org/

8. Malawski, M., Figiela, K., Gajek, A., Zima, A., Bubak, M. (2017). _Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions_. Future Generation Computer Systems, 110, 502–514. https://doi.org/10.1016/j.future.2017.10.029

9. C. Zheng, B. Tovar and D. Thain, "Deploying High Throughput Scientific Workflows on Container Schedulers with Makeflow and Mesos," 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), Madrid, Spain, 2017, pp. 130-139, doi: 10.1109/CCGRID.2017.9. https://ieeexplore.ieee.org/abstract/document/7973696

10. SCAR. https://github.com/grycap/scar

11. Serverless Workflow Specification. (2023). https://serverlessworkflow.io/

12. Synapse Github Repository. https://github.com/serverlessworkflow/synapse

13. Apache SonataFlow Guide. https://sonataflow.org/serverlessworkflow/latest/index.html

14. AWS Step Functions. https://aws.amazon.com/step-functions/

15. Apache Airflow. https://airflow.apache.org/

16. Apache Airflow Architecture Overview. https://airflow.apache.org/docs/apache-airflow/2.6.0/core-concepts/overview.html

17. Temporal. https://temporal.io/

18. Cadence. https://cadenceworkflow.io/docs/get-started

19. Cadence Topology. https://cadenceworkflow.io/docs/concepts/topology

20. Prefect. https://docs.prefect.io/v3/get-started

21. Argo Workflows GitHub Repository. https://github.com/argoproj/argo-workflows

22. Hawaou et al., Industry 4.0 and industrial workflow scheduling: A survey, Journal of Industrial Information Integration, Vol. 38, 2024. https://www.sciencedirect.com/science/article/pii/S2452414X2300119X

23. Argo Workflows Killercoda Course. https://killercoda.com/argoproj/course/argo-workflows/

24. MinIO Object Storage. https://min.io/docs/minio/kubernetes/upstream/index.html

25. MinIO Client. https://min.io/docs/minio/linux/reference/minio-mc.html

26. OSCAR API. https://docs.oscar.grycap.net/latest/api/

27. OSCAR Dashboard. https://github.com/grycap/oscar-dashboard

28. OSCAR Docs. https://docs.oscar.grycap.net/

29. Kubernetes IN Docker. https://kind.sigs.k8s.io/

30. Argo v3.6.7. https://github.com/argoproj/argo-workflows/releases/tag/v3.6.7

31. CLUES. https://github.com/grycap/clues

32. Knative. https://knative.dev/docs/concepts/

33. Kubernetes Namespaces. https://kubernetes.io/es/docs/concepts/overview/working-with-objects/namespaces/

34. Kubernetes Ingress. https://kubernetes.io/docs/concepts/services-networking/ingress/

35. J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," in Proceedings of the IEEE, vol. 63, no. 9, pp. 1278-1308, Sept. 1975, doi: 10.1109/PROC.1975.9939. https://ieeexplore.ieee.org/document/1451869

36. Repositorio smill3r/OSCAR-Argo-Workflows. https://github.com/smill3r/OSCAR-Argo-Workflows

37. Kubernetes Secret. https://kubernetes.io/es/docs/concepts/configuration/secret/

38. Docker Image with Minio Client. https://hub.docker.com/r/bitnami/minio-client/

39. Argo CLI. https://argo-workflows.readthedocs.io/en/latest/walk-through/argo-cli/

40. Argo Installation Documentation. https://argo-workflows.readthedocs.io/en/stable/quick-start/

41. OSCAR Deployment with IM. https://docs.oscar.grycap.net/deploy-im-dashboard/

42. EGI Check In. https://www.egi.eu/service/check-in/

43. Argo Workflows SSO. https://argo-workflows.readthedocs.io/en/latest/argo-server-sso/

44. OSCAR Examples: Cowsay. https://github.com/grycap/oscar/blob/master/examples/cowsay/README.md

45. Argo Asynchronous Job Pattern. https://argo-workflows.readthedocs.io/en/latest/async-pattern/

46. Argo Events. https://argoproj.github.io/events/

47. Argo Worfklows: NodeFieldSelector. https://argo-workflows.readthedocs.io/en/latest/node-field-selector/

48. Grayify Dogs Workflow. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/035258e5712deef27ee22a9c5c2e21bd3e40c023/workflows/image-processing/grayify-dogs.yaml

49. Copy Files WorkflowTemplate. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/035258e5712deef27ee22a9c5c2e21bd3e40c023/workflows/image-processing/templates/copy-files.yaml

50. Wait For Files WorkflowTemplate. https://github.com/smill3r/OSCAR-Argo-

Workflows/blob/035258e5712deef27ee22a9c5c2e21bd3e40c023/workflows/image-processing/templates/wait-for-files.yaml

51. Theano Plant Classification Example. https://github.com/grycap/oscar/blob/master/examples/plant-classification-theano/README.md

52. Classify Plants Workflow. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/6db71a718aa1d39a0530cddf0e0d12aa14d63084/workflows/image-processing/classify-plants.yaml

53. Argo Workflows API. https://argo-workflows.readthedocs.io/en/latest/swagger/

54. Apiumhub. Patrón fan-out/fan-in en Azure Durable Functions. https://apiumhub.com/es/tech-blog-barcelona/azure-patron-fan-out-fan-in/

55. OSCAR Examples: Image Conversion to Grayscale: https://github.com/grycap/oscar/tree/master/examples/imagemagick

56. ImageMagick. https://imagemagick.org/index.php

57. Argo Workflows: Workflow Templates. https://argo-workflows.readthedocs.io/en/latest/workflow-templates/

58. PRAW docs. https://praw.readthedocs.io/en/stable/

59. RoBERTA. https://huggingface.co/docs/transformers/en/model_doc/roberta

60. Transformers. https://pypi.org/project/transformers/

61. Torch. https://pypi.org/project/torch/

62. Repositorio Dashboard Sentiment Analysis. https://github.com/o4tmeal/sentiment-analysis-dashboard

63. Streamlit Cloud. https://streamlit.io/cloud

64. Workflow de Análisis de Sentimiento. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/main/workflows/sentiment-analysis/sentiment-analysis-workflow-dag-notify.yaml

65. Repositorio Workflow Sentiment Analysis. https://github.com/smill3r/OSCAR-Argo-Workflows/tree/main/workflows/sentiment-analysis

66. Script de recolección de datos. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/main/workflows/sentiment-analysis/src/data-collection/recolectar_datos.py

67. Manifiesto FDL para el servicio reddit-collector. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/main/workflows/sentiment-analysis/src/data-collection/recolectar_datos.yaml

68. Collect Data WorkflowTemplate. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/main/workflows/sentiment-analysis/src/data-collection/recolectar_datos.yaml

69. Script que divide el archivo CSV en partes. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/main/workflows/sentiment-analysis/src/split-minio/split-csv.py

70. Manifiesto FDL para el servicio split-csv-sync. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/main/workflows/sentiment-analysis/src/split-minio/split-csv.yaml

71. WorkflowTemplate del paso de Dividir Archivo. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/main/workflows/sentiment-analysis/templates/split-data.yaml

72. Script de análisis de sentimientos. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/main/workflows/sentiment-analysis/src/sentiment-analysis-roberta-notify/sentiment-analysis.py

73. Manifiesto FDL del servicio sentiment-analysis-roberta-notify. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/main/workflows/sentiment-analysis/src/sentiment-analysis-roberta-notify/sentiment-analysis-roberta.yaml

74. WorkflowTemplate del paso de Análisis de Sentimiento. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/main/workflows/sentiment-analysis/templates/sentiment-analysis-notify.yaml

75. Script para unir los archivos resultantes. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/main/workflows/sentiment-analysis/src/join-csv/join-csv.py

76. FDL del servicio join-csv-sync. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/main/workflows/sentiment-analysis/src/join-csv/join-csv.yaml

77. WorkflowTemplate del paso de unir archivos. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/main/workflows/sentiment-analysis/templates/join-data.yaml

78. WorkflowTemplate del paso de descargar resultado. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/main/workflows/sentiment-analysis/templates/download-result.yaml

79. WorkflowTemplate del paso de subir a github. https://github.com/smill3r/OSCAR-Argo-Workflows/blob/main/workflows/sentiment-analysis/templates/commit-to-github.yaml

80. Argo Workflows: Exit Handlers. https://argo-workflows.readthedocs.io/en/latest/walk-through/exit-handlers/