UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Estrategias de Mitigación de Arranque en Frío en Plataformas Serverless On-Premises

Trabajo Fin de Máster

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

AUTOR/A: Quesada Ramírez, Luis Felipe

Tutor/a: Moltó Martínez, Germán

CURSO ACADÉMICO: 2022/2023

Agradecimientos

El autor desea agradecer al profesor Germán Molto Martínez por su retroalimentación tanto en la preparación de este manuscrito como en el desarrollo de cada una de las partes del proyecto. Extender también este agradecimiento a los profesores del máster en su conjunto, por su dedicación y disposición a compartir sus conocimientos con el alumnado.

Quisiera expresar mi más profundo agradecimiento a todas aquellas personas que me han apoyado a lo largo de este proceso para la realización de esta tesis. En primer lugar, agradezco a mi familia por brindarme su amor y apoyo incondicional durante todo este proceso. A las personas que me han acompañado en este camino lleno de retos. Gracias por su compañía y palabras de aliento.

Finalmente, doy gracias a esta prestigiosa institución educativa por acogerme durante este año lectivo. Llevo conmigo invaluables experiencias y conocimiento.

Tabla de Contenidos

Índice de Figuras

Resumen

El siguiente estudio aborda la mitigación del arranque en frío en entornos en la nube y sistemas *serverless on-premises*, centrándose en la técnica de cacheo de imágenes de contenedores Docker en Kubernetes. El uso de la caché en sistemas *serverless* acelera el inicio de aplicaciones al utilizar imágenes previamente descargadas, de modo que se evitan descargas innecesarias en entornos en la nube, y especialmente en los sistemas *serverless* donde los tiempos de respuesta son críticos. La combinación de estas técnicas: cacheo de imágenes Kubernetes y su integración con la arquitectura *serverless*, ofrecen nuevas oportunidades para desarrollar aplicaciones eficientes y altamente escalables en la nube. Para ello, se ha desarrollado la evaluación experimental de dos herramientas de cacheo de imágenes (Kube Fledged y kube-image-keeper) en un entorno *serverless* real utilizando OSCAR. Además de brindar evidencia preliminar de que la integración de herramientas de cacheo de imágenes con arquitecturas serverless como OSCAR puede mejorar el rendimiento y escalabilidad.

Palabras clave: Sistemas *serverless*, Arranque en frío, Nube, Kubernetes, Cacheo de imágenes

Abstract

This study focuses on mitigating cold starts in cloud computing environments by leveraging image caching in Kubernetes and its relationship with *on-premises serverless* systems. Image caching accelerates application startup by storing pre-downloaded container images, eliminating repetitive downloads. In *serverless* environments, where instant response is crucial, image caching becomes even more relevant. By combining these strategies, faster response times and efficient scalability are achieved in *serverless* applications. Image caching in Kubernetes and its integration with the *serverless* architecture provides new opportunities for developing highly efficient and scalable applications in the cloud.

Keywords: *Serverless* systems, Cold start, Cloud, Kubernetes, Image caching

Capítulo 1 - Introducción

En los entornos actuales de desarrollo es común observar el uso de plataformas como Kubernetes en combinación con diferentes paradigmas de computación. Uno de los paradigmas que se está volviendo más común es el uso de Kubernetes junto a la computación serverless.

Antes de entrar en detalles, se procederá a definir varios conceptos importantes y que serán referenciados en el trabajo:

- **Evento**: es una ocurrencia o suceso que sucede en un determinado momento. Por ejemplo, la llegada de un archivo a un bucket de almacenamiento, la solicitud a un endpoint HTTP, o la escritura de datos en una base de datos, son eventos comunes que pueden activar funciones en un sistema serverless [32].
- **Función**: es un fragmento de código encapsulado que se ejecuta para cumplir una tarea específica en respuesta a un evento. Las funciones serverless se ejecutan a demanda y se escalan automáticamente [33].
- **Envío de eventos a funciones**: se refiere al proceso mediante el cual los eventos que ocurren en un sistema (por ejemplo, la llegada de un archivo) desencadenan la ejecución de una función serverless asociada a ese evento. Esto permite responder y procesar los eventos de manera rápida y escalable [34].
- ***Cold Start* o arranque en frío**: se define como el periodo de tiempo que transcurre desde que se invoca una función serverless hasta que realmente se ejecuta. Esto se debe a que primero se debe hacer el aprovisionamiento de toda la infraestructura para que se pueda realizar dicha ejecución [35].

Un ejemplo de lo anterior es la herramienta a utilizar en este trabajo, OSCAR [6], una herramienta desarrollada por el grupo de investigación GRyCAP de Universitat Politècnica de València que permite el procesamiento de datos dirigido a eventos dirigido por eventos mediante un marco de trabajo *serverless* junto a un clúster Kubernetes elástico que puede desplegarse de forma automática en múltiples proveedores Cloud.

Este último, al no contar con infraestructura propia predispuesta para su ejecución, sino más bien dinámica, enfrenta retos que deben ser superados para una óptima ejecución.

Uno de estos retos se conoce como *Cold Start* o arranque en frío [35]. concepto que se define como el periodo de tiempo que transcurre desde que se invoca una función

serverless hasta que realmente se ejecuta, esto debido a que primero se debe hacer el aprovisionamiento de toda la infraestructura para que se pueda realizar dicha ejecución. En la siguiente imagen se presenta un diagrama de flujo de cómo funcionaría un sistema con un sistema de cacheo de imágenes.

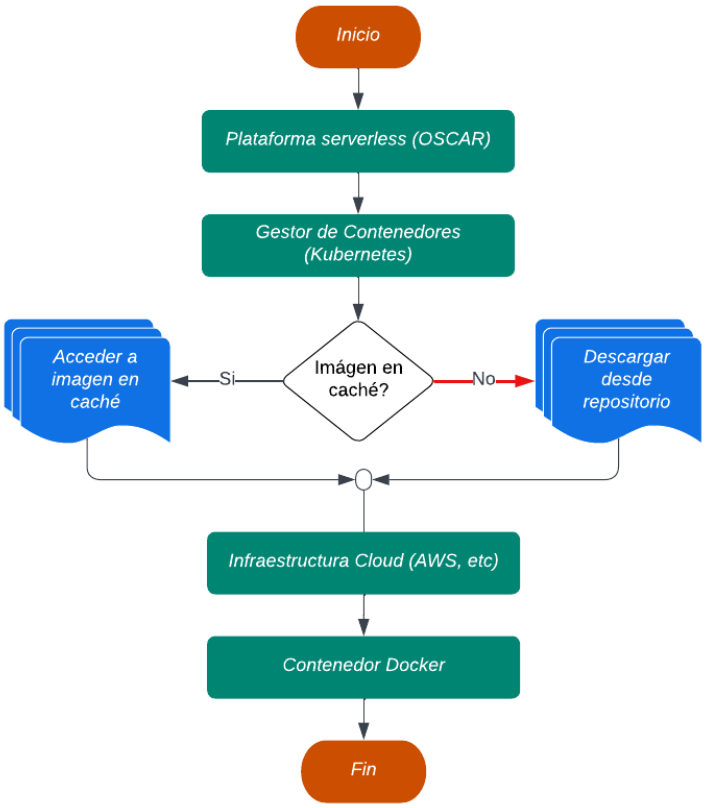


Figura 1 - Flujo de ejecución al usar caché.

Es por esto, que tanto proveedores de servicios en la nube como científicos de la computación y empresas, han desarrollado y probado diferentes técnicas con el objetivo de gestionar los arranques en frío de una mejor manera.

Entre las estrategias utilizadas, se pueden mencionar las que se realizan por medio de software, como lo sería el envío de eventos a la función con el fin de mantener el contenedor listo para cuando sea requerido y que no sea eliminado de memoria. Otra estrategia consiste en el cacheo de imágenes, en especial cuando se utilizan en entornos como Kubernetes, de modo que al ser requerido, solo se debe acudir a la caché en lugar de proceder a su descarga desde un repositorio remoto de imágenes de contenedores, como puede ser Docker Hub o GitHub Container Registry.

Al combinar la utilización de cachés para imágenes con la arquitectura *serverless*, se logra un tiempo de respuesta más rápido, ya que las imágenes están disponibles en memoria o en un almacenamiento eficiente, reduciendo significativamente el tiempo necesario para descargar, extraer y preparar las imágenes de contenedores, lo que se traduce en un arranque más rápido de las funciones *serverless* y una mejora en el rendimiento general de la aplicación.

En este caso específico, se trabajará con OSCAR, aplicación basada en OpenFaaS® por un framework de código abierto que involucra el uso de Kubernetes® y Knative®, entre otras herramientas, para facilitar la ejecución de aplicaciones *serveless* de procesado de datos dirigido por eventos.

En resumen, el cacheo de imágenes desempeña un papel vital en la mitigación del *cold start* en entornos *serverless*. Al almacenar las imágenes de contenedores en caché, se logra un arranque rápido y eficiente de las funciones serverless, lo que permite una respuesta instantánea y una escalabilidad ágil. La combinación de estrategias de cacheo de imágenes y arquitecturas serverless abre nuevas oportunidades para desarrollar aplicaciones más eficientes y altamente escalables en la nube.

1.1 Objetivos

Seguidamente, se presentarán los objetivos perseguidos con este trabajo:

General

- Evaluar herramientas de caché de imágenes de contenedores para clústers OSCAR.

Específicos

- Identificar estrategias utilizadas en la gestión del problema del arranque en frío.
- Conocer el estado del arte con respecto al uso actual de las estrategias relacionadas con el uso de cachés para mitigar el arranque en frío.
- Implementar una solución de caché de imágenes de contenedores Docker dentro de clusters Kubernetes para usar junto a la herramienta OSCAR. de contenedores Docker dentro de clusters Kubernetes

1.2 Metodología

1. **Revisión bibliográfica:** Realizar una revisión de la literatura existente sobre la programación en la nube serverless, las estrategias empleadas en la gestión del

problema del arranque en frío y el estado actual de la utilización de cachés para mitigar este problema. De este modo se obtendrá una comprensión sobre los conceptos y estrategias del tema en cuestión.

2. **Experimento OSCAR - Herramienta de Caché de imágenes**:
   En esta etapa se evaluará el comportamiento de la herramienta de caché junto a un clúster OSCAR.

   a. **Implementación y configuración del entorno**: Configurar el entorno de prueba utilizando un clúster OSCAR y desplegar la herramienta para caché en dicho entorno.

   b. **Ejecución del experimento**: Ejecutar el experimento diseñado previamente, ejecutando pruebas y recolectando datos relevantes. Medir el rendimiento en términos de reducción de arranque en frío y otros aspectos importantes.

3. **Análisis de resultados**: Analizar los datos recolectados durante el experimento y evaluar el comportamiento de la herramienta en el clúster OSCAR. Comparar los resultados con los objetivos planteados y las estrategias estudiadas en la revisión bibliográfica.

4. **Conclusiones y recomendaciones**: Extracción de conclusiones basadas en los resultados obtenidos y brindar recomendaciones para futuras investigaciones en este campo. Resumir los hallazgos clave y destacar las contribuciones de la tesis.

Capítulo 2 - Estado del arte y tecnologías relacionadas

En el estado del arte, se explorarán los conceptos de arranque en frío, cacheo de imágenes, así como conocer herramientas actualmente disponibles que permiten implementar el cacheo de imágenes.

2.1 Docker®

Docker es una plataforma abierta para desarrollar, enviar y ejecutar aplicaciones utilizando contenedores. Algunos conceptos clave sobre Docker:

- **Contenedores**: unidades estandarizadas de software que empaquetan código y dependencias para que las aplicaciones puedan ejecutarse de manera rápida y confiable [40].

- **Imágenes**: plantillas de sólo lectura utilizadas para crear contenedores. Las imágenes se componen de cambios en el sistema de archivos e instrucciones sobre cómo iniciar el contenedor [39].

- **Registros**: repositorios públicos o privados para almacenar y distribuir imágenes de contenedores. Docker Hub es el registro público predeterminado [40].

- **Dockerfile**: archivo de texto con instrucciones para crear una imagen de Docker. Se utiliza para automatizar y documentar cómo se construye una imagen [40].

- **Capas**: Docker crea imágenes a través de una serie de capas. Las capas permiten la reutilización y optimizaciones como el almacenamiento en caché [39].

2.2 Kubernetes®

Es un sistema de orquestación de contenedores de código abierto para automatizar la implementación, el escalado y la gestión de aplicaciones en contenedores. Agrupa contenedores en unidades lógicas para una fácil gestión [38].

Al hablar de kubernetes es importante tener en presentes lo siguientes conceptos:

- **Contenedor**: una unidad estandarizada de software que empaqueta código y dependencias para que la aplicación pueda ejecutarse de manera rápida y confiable. Kubernetes gestiona contenedores que se implementan en pods [38].
- **Nodo**: una máquina de trabajo en el clúster de Kubernetes que ejecuta aplicaciones en contenedores. Cada nodo tiene los servicios necesarios para ejecutar pods y es administrado por los componentes maestros [38].
- **Pod**: la unidad desplegable más pequeña de Kubernetes. Un pod encapsula uno o varios contenedores que comparten recursos como volúmenes y direcciones IP. Los pods son la unidad atómica de la plataforma Kubernetes [38].
- **Clúster**: el conjunto de nodos administrados por Kubernetes para ejecutar aplicaciones en contenedores. Consta de al menos un maestro y varios nodos de cálculo [38].
- ***Control plane* o maestro**: gestiona el clúster de Kubernetes. Incluye API, programadores y controladores que administran cargas de trabajo y se comunican con los nodos [38].
- **CRD (*CustomResourceDefinition*)**: Un mecanismo de extensión API en Kubernetes que permite definir recursos personalizados. Los CRD pueden ampliar Kubernetes con información personalizada y habilitar objetos más allá de los integrados [38].
- **Controladores**: los controladores de Kubernetes monitorean el estado de varios componentes y trabajan para llevar el clúster al estado deseado. Proporcionan funciones automatizadas como escalado automático, actualizaciones continuas, etc [38].
- **Servicio**: una abstracción que define un conjunto lógico de pods y permite la exposición al tráfico externo, el equilibrio de carga y el descubrimiento de servicios para la aplicación que se ejecuta en los pods [38].
- ***ReplicationSet***: un controlador de Kubernetes que garantiza que se esté ejecutando una cantidad específica de réplicas de pod en un momento dado. Se utiliza para garantizar la disponibilidad de un conjunto de pods idénticos [38].
- ***ConfigMap***: un objeto API utilizado para almacenar datos no confidenciales en pares clave-valor. Los pods pueden consumir ConfigMaps como variables de entorno, argumentos de línea de comandos o como archivos de configuración en un volumen. Permite la separación de la configuración específica del entorno de las imágenes del contenedor [38].
- **Espacios de nombres (*Namespaces*)**: un clúster virtual respaldado por el mismo clúster físico. Los espacios de nombres proporcionan aislamiento para equipos, aplicaciones y entornos en un clúster. Los recursos deben ser únicos dentro de un espacio de nombres [38].
- **Etiquetas (*Labels*)**: pares clave-valor adjuntos a objetos de Kubernetes, como pods, implementaciones, etc. Las etiquetas se utilizan para seleccionar

subconjuntos de objetos que satisfacen ciertas condiciones. Ayudan a identificar atributos de los objetos [38].

- **Anotaciones (*Annotations*)**: metadatos no identificativos de objetos. Se utilizan para registrar otros detalles como la versión de compilación, información de contacto, etc. Las anotaciones no identifican objetos y no se pueden utilizar para realizar selecciones [38].
- ***Webhook* de modificación**: En el contexto de Kubernetes, corresponde a una rutina que se ejecuta antes de que una petición sea enviada al servidor de *API* de Kubernetes con las modificaciones requeridas sobre la petición [41]

2.3 *Serverless Computing*

Serverless computing es un paradigma de computación que involucra el uso de servicios gestionados por un proveedor, que se encarga del aprovisionamiento dinámico de recursos sobre la infraestructura de cómputo subyacente para ofrecer elasticidad y robustez automática. Mediante este enfoque, y siguiendo el modelo FaaS (*Functions as a Service*), el programador se enfoca en dividir y desarrollar el sistema en funciones o pequeñas unidades de desarrollo.

Ejemplos de servicios serverless que siguen el modelo FaaS son AWS Lambda, Microsoft Azure Functions o Google Cloud Functions. A modo de ejemplo, AWS Lambda permite:

- Ejecutar código en respuesta a eventos como cambios en un bucket S3, solicitudes HTTP, actualizaciones de bases de datos, etc [36].
- Ejecutar código sin tener que administrar servidores. AWS Lambda se encarga de la disponibilidad y escalabilidad [36].
- Pagar sólo por el tiempo de cómputo consumido. No hay cargos cuando la función no se está ejecutando [36].
- Escalar automáticamente las ejecuciones de la función para atender picos de tráfico [36].
- Integrarse fácilmente con otros servicios de AWS como DynamoDB, S3, SNS, etc [36].
- Desplegar funciones en varios lenguajes como Node.js, Python, Java, C# o Go [36].
- Tener tiempos de inicio muy rápidos, del orden de milisegundos, ideal para trabajo basado en eventos [36].

Como ventajas de este modelo se pueden mencionar la reducción en costos, así como un modelo simplificado para desarrollo de aplicaciones en la nube; el operador tiene la oportunidad de gestionar los pila de desarrollo y ofrecer una plataforma que permita la utilización de otros servicios ofrecidos y la reducción del esfuerzo para crear aplicaciones.

Dentro de las desventajas se puede mencionar la dependencia de las decisiones del proveedor en cuanto a la calidad del servicio, escalado y tolerancia a fallos.

Así como se presentan ventajas y desventajas, también se presentan desafíos; uno de los más comunes y que ha sido objeto de estudio por diversos científicos de la computación es el *Cold Start* o Arranque en Frío.

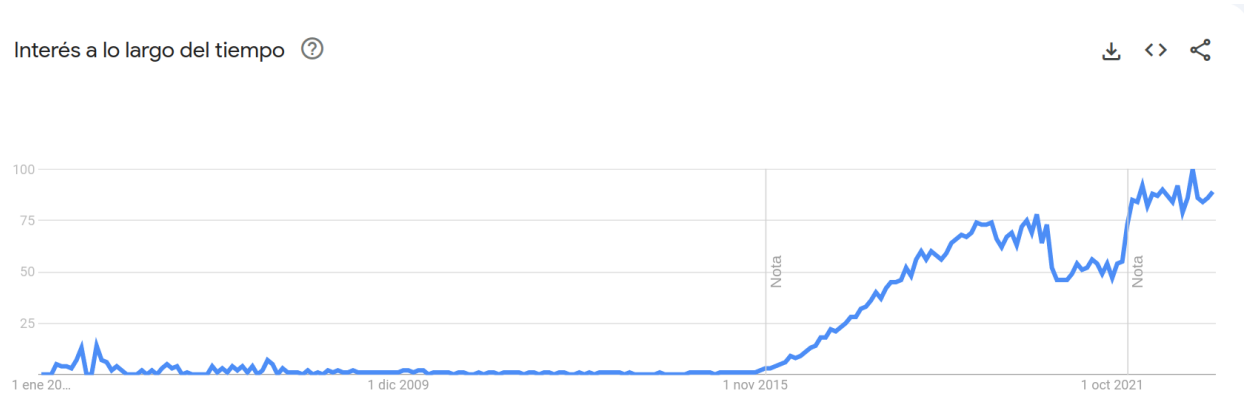


Figura 2 - Tendencia de búsqueda del término Serverless Computing (Google Trends). Fuente [5].

2.4 Arranque en frío (*Cold Start*)

Como parte del creciente uso del paradigma de desarrollo serverless, el arranque en frío se ha convertido en un tema de estudio.

A continuación se presenta una imagen que explica el fenómeno del arranque en frío:

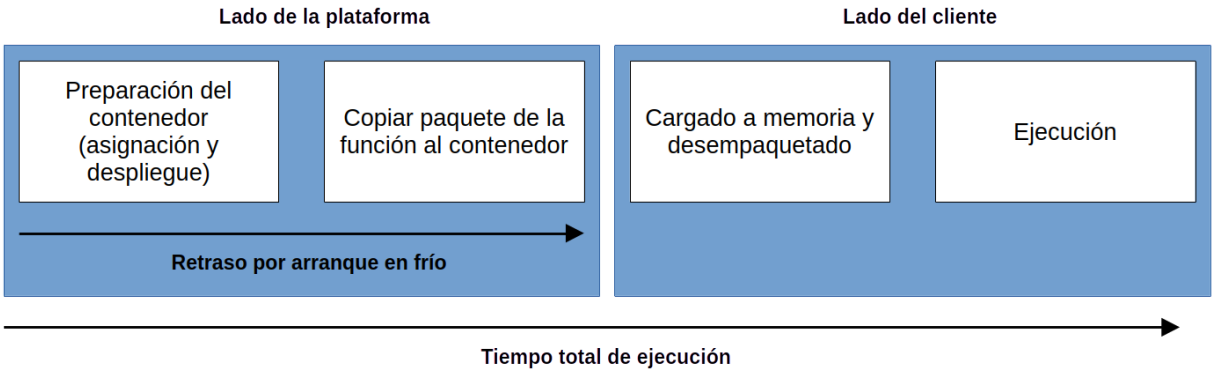


Figura 3 - Pasos para la ejecución de una función serverless. Fuente [42].

Es así como se han empezado a identificar características de las funciones serverless y de infraestructura que, si se implementan, permiten la reducción de los tiempos requeridos para la ejecución de las funciones.

Se han detectado algunas mejores prácticas de programación que pueden ayudar a mitigar el arranque en frío en este tipo de sistemas. Algunos puntos a tener en cuenta son los siguientes: [1], [20]

- **Evitar funciones muy grandes**: Funciones con gran cantidad de código o numerosas dependencias requieren de tiempos mayores para responder a su invocación, ya que requieren más tiempo para cargar todas las dependencias y el código, antes de su ejecución.

- **Reducir las variables de configuración**: Esto es aplicable a lenguajes estáticos como C# o Java, puesto que se desea evitar variables y clases innecesarias.

- **Ajustar la asignación de memoria**: Si se utilizan servicios en la nube como *AWS, Azure o Google Cloud*, se recomienda ajustar la memoria, ya que se genera una relación directamente proporcional entre este factor y la cantidad de CPU que será asignada a la función. Una cantidad adecuada de CPU disminuye los arranques en frío, a costa de incrementar el coste de cada ejecución.

- **Mantener las funciones precalentadas**: Esta técnica implica realizar invocaciones periódicas a la función *serverless* de modo que se simula actividad perpetua. Para este caso, se cuenta con herramientas como *Lambda Warmer* [21].

- **Reducir el tamaño del paquete**: Solo se deben agregar los archivos requeridos y evitar todo lo que no es necesario para la función serverless.

- **Entorno de ejecución adecuado**: Python presenta el menor tiempo de arranque en frío, mientras que C# y Java presentan una diferencia de casi 100 veces más para estos lenguajes.

Otras técnicas que se pueden implementar son las que requieren de configuraciones en la infraestructura o de algún tipo de software adicional para evitar que los recursos se liberen. Como menciona *Vahidinia et al.* [5], hay dos tipos de estrategias que se pueden seguir: 1) optimización del ambiente de ejecución y 2) Reducir el retraso por carga de librerías.

1. Optimización del entorno de ejecución

Para lograr una optimización adecuada, se pueden seguir los siguientes enfoques:

- Mantener un contenedor por función, siempre en ejecución.
- Si la plataforma *serverless* que se está utilizando lo permite, pausar el contenedor mientras no sea requerido.
- Emplear la técnica *pre-baking* o pre-horneado, que consiste en mantener una serie de contenedores listos con todas las dependencias requeridas para cuando sean requeridos.
- Mantener los contenedores tibios después de la primera ejecución. Por contenedor tibio se entiende aquel que ha sido ejecutado recientemente y que todavía reside en la memoria del sistema, permitiendo un inicio más acelerado para la atención de peticiones [37].
- Hacer uso de colas, una para contenedores calientes y otra para los contenedores fríos. Un contenedor caliente es el que se encuentra ejecutándose constantemente para que la atención de futuras peticiones se realice de manera más rápida. Un contenedor frío se refiere a aquel que no se ha ejecutado, y por ende, requiere más tiempo de inicio [37].

2. Reducir el retraso por carga de librerías

Las librerías requeridas se encuentran pre-cargadas, instaladas y almacenadas en memoria.

3. Carga perezosa de imágenes

Solo se cargan las dependencias requeridas para que el contenedor se inicialice. Según Harter et al. [25], solo el 6% de una imagen Docker es requerida para iniciar un contenedor.

4. **Enfoque de colas**

Se busca inicializar la infraestructura necesaria de manera anticipada para así crear un conjunto de recursos que se encuentren listos para ejecutarse cuando se requiera. Según los resultados arrojados, se logró la reducción del arranque en frío en un 85%. Lin et al. [26]

5. **Uso de caché.**

Actualmente, utilizar una caché para almacenar imágenes es una opción para hacerle frente al problema del arranque en frío, razón por la cual, se pueden encontrar herramientas que permiten mantener las imágenes en caché, de modo que no sea requerida la descarga desde un repositorio remoto.

Otro enfoque en la utilización de cachés, lo menciona Heydari et al. [24], en el cual se propone el empleo de un planificador de tareas, el cual se encarga de generar imágenes en aquellos nodos donde se encuentra la imagen o una capa de la misma ya disponible.

2.4.1 Soluciones de proveedores de nube pública

Los proveedores de servicios en la nube también han tratado de resolver este problema. A continuación se presentan algunas de sus estrategias o herramientas a disposición de los usuarios para mitigar los efectos del arranque en frío.

Alibaba Cloud™

La solución ofrecida por este proveedor utiliza la capacidad extensible de Kubernetes mediante los *CustomResourceDefinition* (CRD) y define el recurso *ImageCache* permitiendo de este modo la creación de *pods* de una manera más acelerada [11].

Permite la selección de imágenes por medio de anotaciones Kubernetes y se ofrecen dos tipos de anotaciones para cumplir con la selección de imágenes; la primera anotación permite la selección automática de las imágenes; la segunda opción consiste en explícitamente indicar la imagen que se desea tener en la caché [11].

AWS™

Amazon AWS ofrece *Amazon Elastic Container Container Registry* (Amazon ECR), Mediante este servicio y directivas de Kubernetes como "IfNotPresent" o “Always”, se logra mantener en la caché la imagen requerida; lo anterior permite retener la imagen en caché pero no incluye la descarga previa de la imagen

Otro tipo de estrategia que se puede utilizar, consiste en la técnica llamada *Provisioned Concurrency* o Concurrencia Aprovisionada.

En AWS, es la cantidad de concurrencia (peticiones) que una función puede gestionar en el tiempo y se definen dos tipos; 1) la reservada y 2) la aprovisionada. En la reservada se define el número máximo de instancias asignadas a la función. En la aprovisionada, se puede definir un número definido de ambientes (por ambientes se puede entender memoria, cpu, red, disco, etc) que desea asignar a una función. Los entornos de ejecución se preparan para responder de manera inmediata a las solicitudes entrantes [14], [15].

Microsoft Azure™

Como menciona (Bekker, 2019) [12] y (Lasker, 2019) [13], *Microsoft Azure* ofrece el proyecto *Teleport* [12], el cual se encuentra todavía en fase de pruebas y que busca eliminar las etapas de descarga y descompresión de imágenes.

La compañía quiere hacer uso de sus redes de alta velocidad, de modo que, al iniciarse un contenedor, en lugar de realizarse la descarga de la imagen, se enviarán puntos de montaje a los cuales el contenedor deberá conectarse según los volúmenes requeridos y así, poder hacer uso de las imágenes de contenedores, sin necesidad de descargarlas. Estos puntos de montaje, referencian lo que en *Azure* se conoce como capas expandidas.

La siguiente imagen, tomada de Lasker [13], permite observar el flujo de *Teleport* y su funcionamiento:

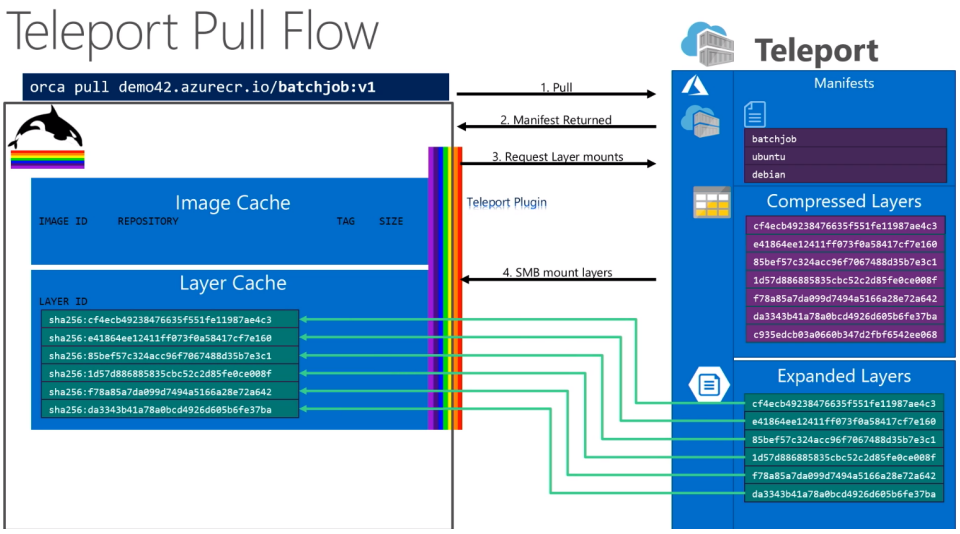


Figura 4 - Flujo de Teleport. Fuente [13].

A continuación, algunas estadísticas del rendimiento mediante este enfoque:

Host Layers	182 MB 6	1.7 GB 7	5.1 GB 34	1.8 K 1
Dedicated VM	12.7	83.9	412.8	1.8
ACI *	66.4	188.1	522.4	25.3
Project Teleport	**3.3**	**4.1**	**7.6**	**2.8**

Figura 5 - Estadísticas de Teleport. Fuente [13].

En la figura anterior se pueden observar las diferencias en tiempos comparando máquinas virtuales dedicadas, *Azure Container Instances (ACI)*, y la tecnología de *Teleportation*. Se puede observar una disminución notable en el tiempo requerido para acceder a las imágenes en un almacenamiento remoto en comparación con los otros enfoques

Google Cloud Platform™
Como se menciona en e[4], Google ha desarrollado el subsistema *Image Streaming*. Este consiste en la transferencia de imágenes según sean requeridas por el contenedor. *Image Streaming*, se conecta a un sistema de archivos remotos, de los cuales se realiza la transmisión de las imágenes. De este modo, los contenedores pueden ser inicializados solo con las dependencias necesarias y de este modo, incrementando los tiempos de espera.

2.4.2 Soluciones de herramientas On-Premises.

El siguiente apartado se centra en herramientas de código abierto para soportar la computación serverless en entornos on-premises, es decir, en los recursos de una organización.

OpenFaaS®.
OpenFaas [43], es una plataforma que permite el despliegue de funciones dirigidas por eventos en un clúster *Kubernetes*. Permite a los desarrolladores empaquetar binarios en imágenes de *Docker*.

Es así como esta herramienta propone diferentes técnicas para afrontar el *cold start*, especialmente se puede mencionar la siguiente: [28]

- *Scale-to-zero*: Monitor que se realiza con el objetivo de determinar el tráfico de una función y eliminar las instancias en estado suspendido.

Esta opción todavía implica un retraso de 1 a 2 segundos, razón por la cual debe acompañarse de otras técnicas como lo son:

- *Pre-pull de imágenes*: esta técnica permite descargar una imagen antes de que sea requerido su uso, minimizando así el tiempo de arranque en frío.
- Optimización de las *readiness probes*: Estas son el componente encargado de determinar cuándo un contenedor dentro del clúster está preparado para recibir peticiones.
- Réplicas tibias: mantener cierto número de instancias de las funciones listas para atender futuras peticiones.

OpenWhisk™

OpenWhisk [44] permite a los desarrolladores la escritura de lógicas funcionales llamadas *Actions* en cualquier lenguaje de programación. OpenWhisk utiliza contenedores y técnicas de escalado para la optimización del tiempo de activación de funciones.

Mediante el uso de tecnologías de contenedores se permite la ejecución de las funciones en un entorno aislado. Específicamente, OpenWhisk mantiene los entornos precalentados, es decir, los mantiene en memoria de modo que se reduce el tiempo del arranque en frío.

Otro método que utiliza OpenWhisk consiste en técnicas de autoescalado, permitiendo la asignación de recursos y utilizando, tomando como base, la carga de trabajo entrante, distribuyendo así, una función entre los nodos del sistema que estén listo para recibir peticiones.

2.4.3 Herramientas para el cacheo de imágenes de contenedores

Existen diversas herramientas que han tratado de resolver este problema mediante diferentes enfoques:

Kube Fledged

Como se menciona en [8] y [9], Kube Fledged es una extensión que permite la creación y gestión de una caché para imágenes cuyo objetivo es 1) permitir el inicio de los *pods* de un modo mucho más ágil y 2) permitir a las funciones *serverless* responder de manera inmediata cuando son invocadas.

Además, Kube Fledged provee *APIs* para gestionar el ciclo de vida de la caché y permite la configuración de parámetros para personalizar el funcionamiento según las necesidades.

Dentro de las principales características de Kube Fledged está su facilidad de uso, ya que funciona como una extensión para el clúster Kubernetes, lo que significa que su implementación en un clúster es poco invasiva.

Casos de uso:

- Aplicaciones que necesitan iniciarse rápidamente. (aplicaciones que realizan el procesamiento de datos en tiempo real deben escalar rápidamente debido a grandes volúmenes de datos).
- Funciones serverless porque necesitan reaccionar inmediatamente a los eventos entrantes.
- Aplicaciones IoT que se ejecutan en dispositivos porque la conexión de red entre el dispositivo perimetral y el repositorio de imágenes es intermitente.
- Si es necesario extraer imágenes de un registro privado y no es posible dar acceso a todos para extraer imágenes de ese registro, estas imágenes pueden estar disponibles en los nodos del clúster.
- Si el administrador u operador del clúster necesita actualizar la aplicación y desea verificar de antemano si la nueva imagen se puede extraer correctamente.

Kube Fledged está creado como una extensión utilizando los recursos personalizados (*Custom Resources*). Es así como define un tipo llamado *ImageCache* e implementa un controlador llamado *kubefledged-controller* el cual se encarga de realizar acciones como la descarga y borrado de las imágenes.

Ofrece acciones *CRUD* sobre la caché, razón por la cual la gestión de la caché se puede realizar mediante el uso de comandos *kubectl*, del mismo modo que se realizaría con otros recursos Kubernetes.

Arquitectura de Kube Fledged

La arquitectura de Kube Fledged se muestra en la siguiente imagen:

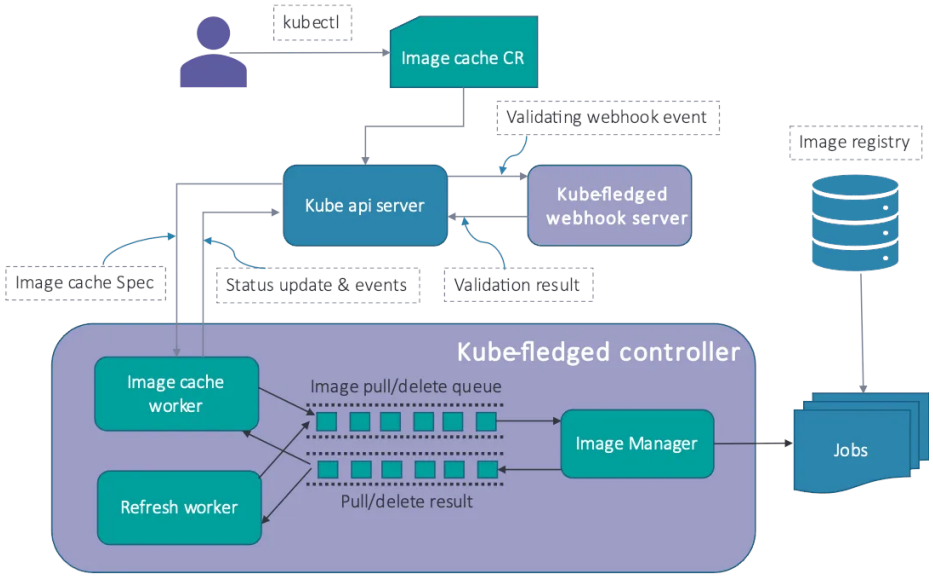


Figura 6 - Arquitectura de Kube Fledged. Fuente [8]

A continuación se comentan cada uno de sus componentes:

kubefledged-controller: controlador personalizado responsable de gestionar la caché de imágenes y ejecuta diversas tareas como la descarga y eliminación de imágenes. Interactúa con el servidor de API de Kubernetes para crear y eliminar recursos.

Image Manager: La rutina encargada de la descarga y eliminación de imágenes. Utiliza jobs de Kubernetes para realizar estas operaciones. Realiza un seguimiento del estado de descarga de imágenes, actualizaciones y eliminaciones y actualiza el campo de estado del recurso ImageCache en consecuencia.

kubefledged-webhook-server: Componente responsable de la validación de los campos incluidos en el manifiesto ImageCache. Recibe eventos de validación cuando se crea o modifica un recurso ImageCache. Asegura que el cacheSpec (Anexo 3) del manifiesto ImageCache sea válido.

ImageCache: Recurso personalizado que representa una caché para almacenar y gestionar imágenes de contenedores. Los usuarios pueden crear un recurso ImageCache especificando la lista de imágenes a descargar y almacenar en la caché, junto con un nodeSelector opcional para especificar los nodos objetivo donde se almacenarán en caché las imágenes.

Refresh Worker: Rutina de trabajo de actualización que se ejecuta periódicamente para mantener actualizada la caché de imágenes. Verifica si alguna imagen falta en la

caché (por ejemplo, eliminada por la recolección de basura de imágenes de kubelet) y las vuelve a descargar en la caché. Se asegura que la caché esté actualizada con las últimas versiones de las imágenes.

Kube Image Keeper (kuik)

Nació en Francia en la empresa Enix, encargada de gestionar infraestructura en la nube y on-premises para diversos clientes. Los siguientes son algunos de los retos a los que se enfrentaba el equipo de gestión y que culmina en el desarrollo de Kube Image Keeper:

1. Registros públicos no disponibles y problemas de rendimiento: Se notó que había momentos en los cuales los registros de imágenes de contenedores públicos estaban inactivos o experimentaban un rendimiento reducido, lo que provocaba que las imágenes de contenedores no se descargaran cuando era requerido.

2. Límites de extracción aplicados por algunos registros: Ciertos registros públicos, especialmente aquellos con cuentas de nivel gratuito, imponen límites a la cantidad de descargas de imágenes de contenedores que se pueden descargar. Dicha limitación puede dificultar las operaciones del clúster, especialmente si varios nodos requieren extraer la misma imagen.

3. Imágenes eliminadas debido a políticas de administración de espacio en disco: En algunos casos, las imágenes de contenedores se eliminan automáticamente de los registros debido a políticas de administración de espacio en disco. Esto podría provocar que los contenedores no se inicien cuando la imagen sea requerida ya que no se encuentra disponible.

4. Independencia de los servicios específicos de la nube: Se requería una solución que funcionara sin problemas tanto en clústeres locales como los basados en la nube sin depender de servicios específicos de la nube o características específicas del centro de datos.

5. Optimización de las llamadas de registro de origen: Las soluciones existentes exploradas no optimizan de manera efectiva las llamadas de registro de origen, lo que provocó un mayor consumo de recursos y posibles problemas de rendimiento.

6. Flexibilidad en la exclusión de pods y personalización de imágenes: La solución necesitaba permitir la exclusión selectiva de ciertos pods o imágenes del sistema de almacenamiento en caché y proporcionar un control granular sobre las configuraciones de almacenamiento en caché.

Una vez entendidas las limitaciones, y una vez probadas las herramientas ya disponibles en el mercado, se procede al desarrollo de Kuik.

En este caso, Kuik está compuesto esencialmente por un *Docker Registry* y un componente personalizado (*operator*), cuya arquitectura describiremos a continuación: [10]

1. Un **webhook** de modificación (ver conceptos de Kubernetes) que crea sobre la marcha los manifiestos de los pods y actualiza el registro que utiliza.
2. Dos **controladores** con funciones diferentes:
    - *Primer controlador*: Examina los pods y crea recursos (CRIs) *CachedImages* según sea necesario.
    - *Segundo controlador*: monitorea los recursos *CachedImages* y se encarga de ubicar en caché las imágenes.
3. Un ***proxy*** que se encarga de recibir las peticiones, y en caso de no contar con la imagen en caché, redirige la solicitud al registro de imágenes principal.

A continuación, se presenta una imagen de la arquitectura de Kube Image Keeper (Kuik):

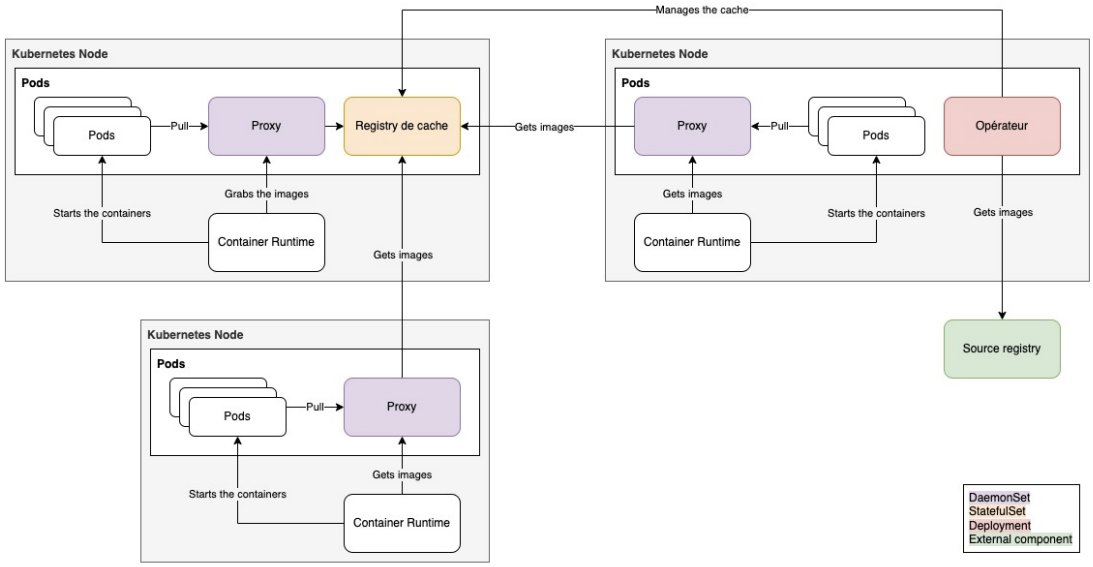


Figura 7 - Arquitectura de Kube Image Keeper. Fuente [10]

Stargz Snapshotter / eStargz

La idea detrás de *Stargz Snapshotter* es permitir la inicialización de un contenedor sin esperar a que se extraiga la imagen completa; solo los fragmentos necesarios de la imagen se recuperan bajo demanda [17].

Las características clave y los detalles de Stargz Snapshotter incluyen:

Formato eStargz: el formato eStargz, que está diseñado para la descarga perezosa de imágenes de contenedores (*lazy pulling)*. Este formato es compatible con imágenes OCI/Docker y se puede enviar a registros de contenedores estándar como GitHub Contaner Registry.

Mejoras de rendimiento: Se utilizaron puntos de referencia para comparar los tiempos de inicio de los contenedores. Los resultados demuestran que el uso de Stargz Snapshotter con imágenes en formato eStargz brinda mejoras de rendimiento en términos de operaciones de extracción, aunque podría haber un ligero inconveniente de rendimiento para las operaciones de ejecución debido a la obtención de archivos bajo demanda.

Integración: el proyecto proporciona instrucciones para integrar Stargz Snapshotter con varias herramientas y plataformas. Esto incluye la integración con Kubernetes, tiempos de ejecución de contenedores como CRI-O y Podman, y systemd.

Rápido inicio: las guías de inicio rápido están disponibles para comenzar con Stargz Snapshotter, tanto con Kubernetes como con la herramienta CLI nerdctl. Las guías incluyen instrucciones para configurar y ejecutar snapshotter, así como para crear pods con imágenes eStargz.

Creación de imágenes: el proyecto ofrece información sobre la creación de imágenes eStargz utilizando diferentes métodos, como Docker Buildx, Kaniko, nerdctl y ctr-remote. Estos métodos permiten crear y optimizar imágenes en formato eStargz para la extracción perezosa.

Funciones experimentales: el proyecto también menciona funciones experimentales, como la ejecución de contenedores en IPFS con extracción perezosa, y proporciona un enlace a la documentación para obtener más detalles.

Guías de uso: se proporcionan ejemplos para ilustrar cómo usar Stargz Snapshotter con Kubernetes, incluidas las configuraciones y los comandos necesarios para crear pods de eStargz.

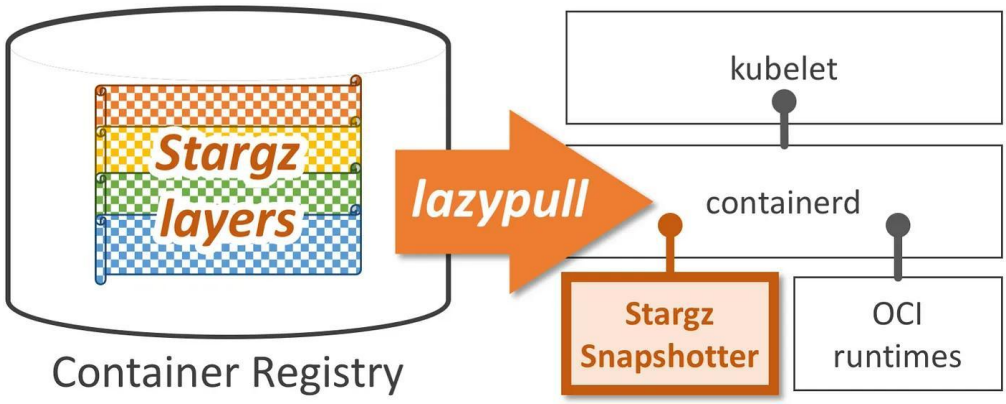


Figura 8 - Esquema de funcionamiento de Stargz Snapshotter. Fuente [17]

Uber Kraken

Kraken es un registro de Docker punto a punto (P2P) de código abierto desarrollado por el equipo de administración de clústeres de Uber para abordar los problemas de rendimiento con su registro de Docker. Este registro de Docker con tecnología P2P está diseñado para centrarse en la escalabilidad y la disponibilidad, abordando específicamente los desafíos en la distribución de imágenes dentro de los clústeres de cómputo a gran escala y los entornos de nube híbrida.

Kraken está diseñado para la administración, replicación y distribución de imágenes de Docker en configuraciones de nube híbrida. Ofrece soporte de back-end y funciona como una extensión que se puede agregar al clúster, lo que le permite integrarse sin problemas con las configuraciones de registro de Docker existentes. Uno de sus principales objetivos es distribuir imágenes de Docker de manera eficiente en entornos con múltiples regiones y sistemas de nube híbrida.

Características clave de Kraken:

- **Altamente escalable**: distribuye imágenes de Docker a más del 50% del límite máximo de velocidad de descarga en cada host.
- Admite al menos 15 000 hosts por clúster.
- **Admite grandes blobs[1]/capas**, con un tamaño máximo típico de 20 GB para un rendimiento óptimo.
- **Altamente disponible**, asegurando que ningún componente individual sea un punto de fallo.

[1] *Binary Large Object (*blob): Es un objeto que almacena una gran cantidad de datos binarios como una sola entidad. Los blobs se utilizan comúnmente para almacenar imágenes, audio y otros objetos multimedia, aunque a veces se almacena código ejecutable binario.

- Proporciona autenticación segura y protección de la integridad de los datos a través de TLS.
- **Admite opciones de almacenamiento conectables**, conectándose a servicios de almacenamiento de blobs confiables como S3, GCS, ECR, HDFS u otros registros.
- **Implementa replicación entre clústeres sin pérdidas**, lo que permite la replicación asincrónica entre clústeres.

La arquitectura de Kraken utiliza un protocolo P2P que implica hosts dedicados que envían contenido a una red de agentes que se ejecutan en cada host del clúster. Un componente central llamado rastreador orquesta esta red, formando un grafo que asegura una alta conectividad y un diámetro pequeño.

La arquitectura incluye los siguientes componentes:

- **Agente**: se ejecuta en cada host, implementa la interfaz de registro de Docker, notifica sobre contenido disponible a un rastreador y se conecta a los pares para la descarga de contenido.
- **Origen**: sembradores dedicados que almacenan blobs como archivos en disco utilizando almacenamiento conectable.
- **Rastreador**: realiza un seguimiento de los pares con contenido, proporciona listas de pares para descargar blobs y organiza la red.
- **Proxy**: implementa la interfaz de registro de Docker, carga capas de imágenes en los orígenes y administra las etiquetas y la replicación.
- **Build-Index**: asigna etiquetas a los blobs, lo que facilita la replicación de imágenes entre clústeres.

La siguiente imagen presenta la arquitectura descrita anteriormente:

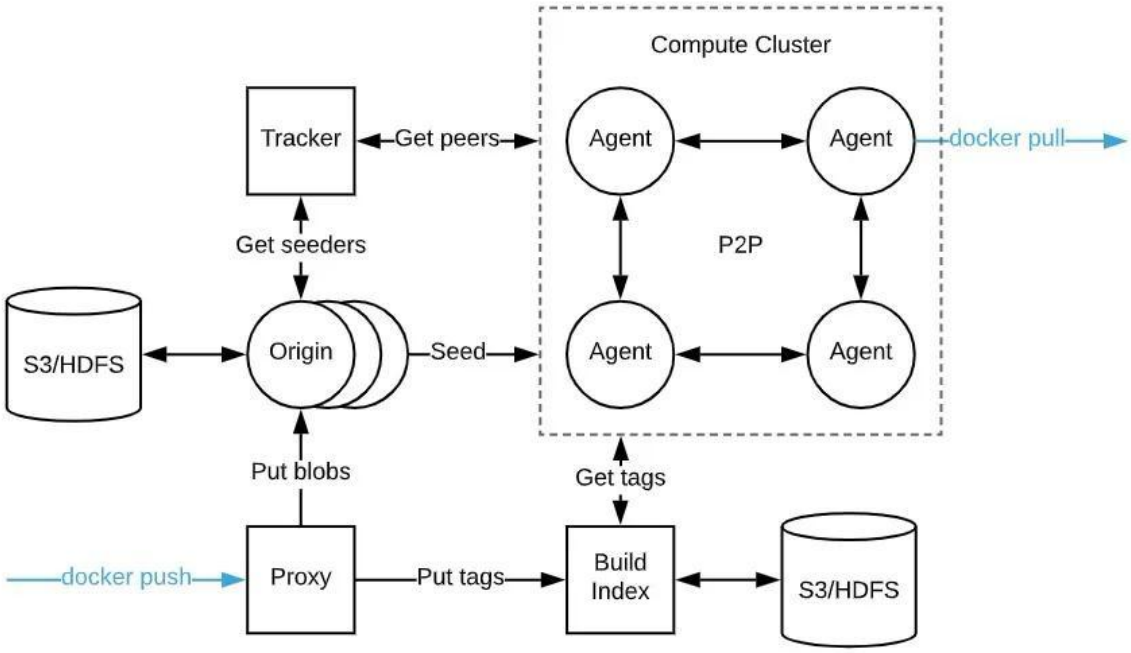


Figura 9 - Esquema de Stargz Snapshotter. Fuente [31]

Los datos de referencia de Kraken muestran su excelente rendimiento. Por ejemplo, puede distribuir más de 1 millón de blobs por día, incluidos 100 000 blobs de más de 1 GB. En el pico de carga de producción, Kraken puede distribuir 20 000 blobs de 100 MB a 1 GB en menos de 30 segundos.

La implementación exitosa de Kraken en Uber desde principios de 2018 y su impacto significativo en la velocidad de distribución de imágenes llevaron a su código abierto. Uber tiene como objetivo compartir esta herramienta con la comunidad de código abierto más amplia para fomentar debates sobre las mejores prácticas en la construcción de una infraestructura Docker adaptable y confiable.

Imagewolf

Es una herramienta que utiliza bittorrent para distribuir rápidamente imágenes en un clúster. Esto puede ayudar a reducir el tiempo que lleva desplegar nuevas imágenes, ya que no es necesario que cada nodo extraiga las imágenes de un registro remoto.

ImageWolf es una prueba de concepto; disponible bajo licencia de código abierto y por ende, de libre acceso.

Estos son algunos de los beneficios de usar ImageWolf:

- Reducción del tiempo de descarga de imágenes.
- Mayor disponibilidad del clúster
- Seguridad de imagen mejorada
- Uso de ancho de banda reducido

ImageWolf escucha nuevas inserciones en un registro. Cuando se envía una imagen, ésta es extraída y distribuida a todos los demás nodos del clúster. De este modo, la imagen está disponible para ejecutarse en cualquier nodo del clúster.

ImageWolf puede integrarse tanto con Docker Hub como con los registros locales.

Comparación de herramientas.

Criterios	Kube Fledged	Kube Image Keeper (Kuik)	Stargz Snapshotter	Uber Kraken	ImageWolf
Propósitos	- Imágenes de caché para un inicio de pod más rápido.	- Almacenamiento en caché. - Distribución de imágenes de contenedores.	- *Lazy-pull* para la inicialización del contenedor.	- Registro Docker P2P.	- Distribución rápida de imágenes.
Características clave	- Gestión de caché. - Uso de APIs Kubernetes. - Facilidad de uso. - Fácil instalación.	- Operador personalizado. - Almacenamiento en caché. - Gestión de imágenes.	- Formato *Lazy-pull*. - Mejoras de rendimiento.	- Arquitectura P2P. - Alta Escalabilidad. - Seguridad.	- Distribución BitTorrent. - Despliegue rápido.
Casos de uso	- Inicio rápido de los Pods. - Funciones serverless. - IoT	- Distribución eficiente de imágenes. - Escalado	- Inicio del contenedor más rápido. - Aumento del rendimiento	- Distribución de imágenes de Docker a gran escala	- Distribución rápida de imágenes. - Disponibilidad

Criterios	Kube Fledged	Kube Image Keeper (Kuik)	Stargz Snapshotter	Uber Kraken	ImageWolf
Integraciones	- Kubernetes. - Custom Resources. - kubectl.	- Docker Hub. - Registros locales.	- Kubernetes. - CRI-O. - Podman. - systemd	- Docker. - Nube híbrida. - Clústeres a gran escala.	- Docker Hub. - Registros locales.
Licencia	- Apache 2.0.	- MIT.	- Apache 2.0.	- Apache 2.0.	- Código abierto. Licencia no especificada
Escalabilidad	- Permite colocar imágenes en el nodo principal y nodos auxiliares.	- Permite colocar imágenes en el nodo principal y nodos auxiliares.	- Escalable, soporta múltiples plataformas.	- Arquitectura P2P altamente escalable.	- Distribución de imágenes P2P escalable.
Beneficios Adicionales	- Rendimiento serverless mejorado. - Compatibilidad con IoT.	- Flexibilidad. - Exclusión de pods.	- Eficiente extracción perezosa. - Beneficios de rendimiento.	- Disponibilidad. - Confiabilidad.	- Tiempo de implementación reducido.

2.5 Tecnologías relacionadas.

2.5.1 OSCAR

Es un marco de trabajo serverless de código abierto, que permite crear aplicaciones de procesamiento de datos. Se ejecuta en un clúster elástico Kubernetes que se despliega utilizando la siguiente herramienta [6]:

- **Infrastructure Manager (IM)**: herramienta de código abierto que permite aprovisionar infraestructura en diferentes nubes.

Los siguientes componentes se despliegan dentro del clúster Kubernetes, para facilitar la ejecución de OSCAR:

- **CLUES**: gestor de elasticidad que permite escalar horizontalmente el clúster Kubernetes de acuerdo a las cargas de trabajo.
- **Minio**: Almacenamiento de alto desempeño distribuido que ofrece una API compatible con S3.
- **Knative**: es un marco serverless para servir aplicaciones basadas en contenedores para invocaciones síncronas.
- **OSCAR Manager**: API principal responsable de la gestión de los servicios y la integración de los diferentes componentes.
- **OSCAR UI**: interfaz de usuario gráfica fácil de usar dirigida a los usuarios finales.

OSCAR también hace uso de los siguientes servicios de almacenamiento:

- **Servidores externos Minio**, que pueden estar en clústeres distintos de la plataforma.
- **Amazon S3**, un servicio de almacenamiento de objetos que ofrece escalabilidad, disponibilidad de datos, seguridad y rendimiento en la nube pública de AWS.
- **Onedata**, la solución global de acceso a datos para la ciencia, utilizada en la Nube Federada EGI.
- **dCache**, un sistema para almacenar y recuperar grandes cantidades de datos, distribuidos entre un gran número de nodos de servidor heterogéneos, bajo un único árbol de sistema de archivos virtual con una variedad de métodos de acceso estándar.

La siguiente imagen muestra los componentes de OSCAR de manera gráfica [6]:

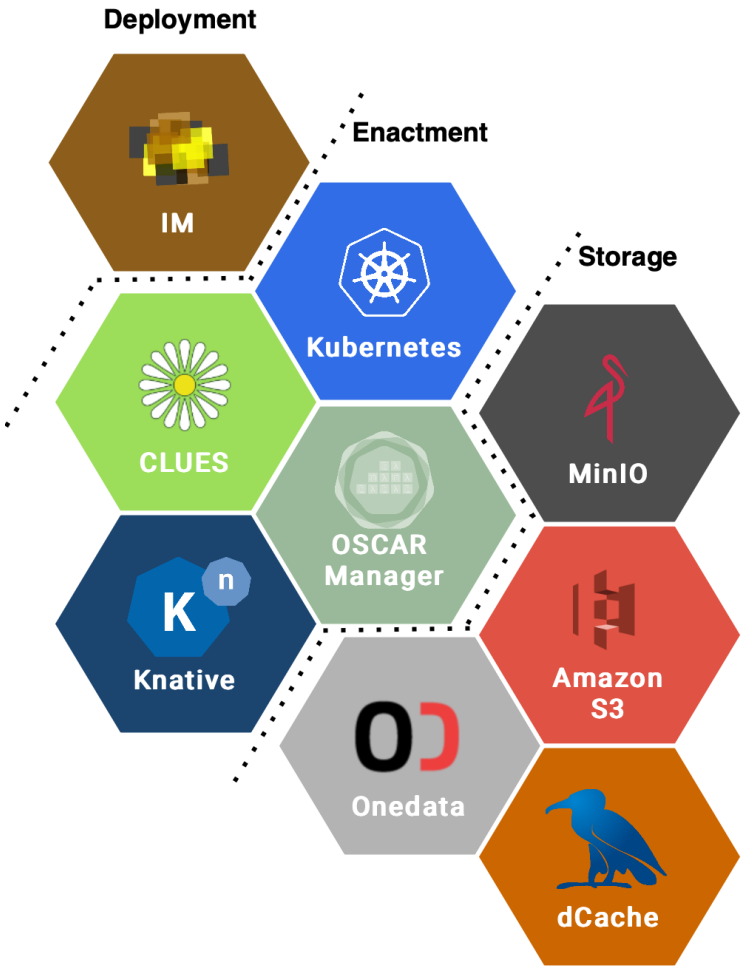


Figura 10 - Componentes de OSCAR. Fuente [6]

OSCAR será la herramienta principal en la cuál se estudiará el impacto de la caché en una plataforma serverless.

2.5.2 OpeLens

OpenLens es el proyecto de código fuente abierto bajo licencia MIT, del cual se origina el escritorio Lens como una rama. Es una herramienta de desarrollo para entornos Kubernetes y proyectos en contenedores. Los pioneros del ecosistema nativo de la nube se unieron y formaron el proyecto, y está disponible en el repositorio de OpenLens en Github.

OpenLens permite la personalización y gestión de los clústeres Kubernetes mediante la instalación de extensiones, de modo que admite agregar diferentes funcionalidades según sea requerido.

La siguiente imagen presenta la pantalla inicial de OpenLens. En este caso, como el clúster cuenta con el paquete de métricas, OpenLens lo detecta y permite la visualización del clúster

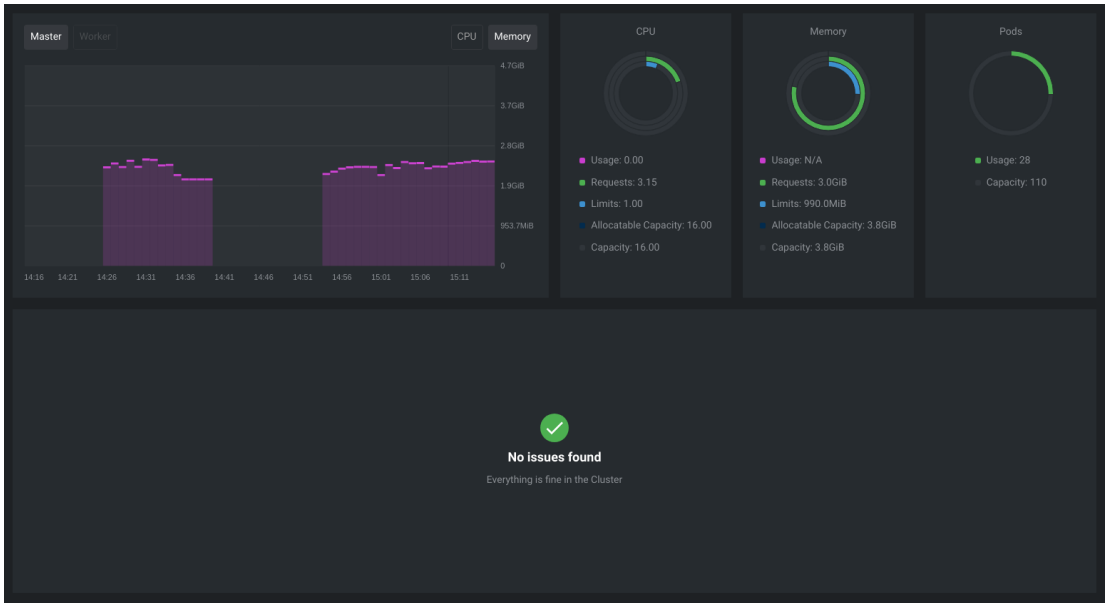


Figura 11 - Dashboard de OpenLens.

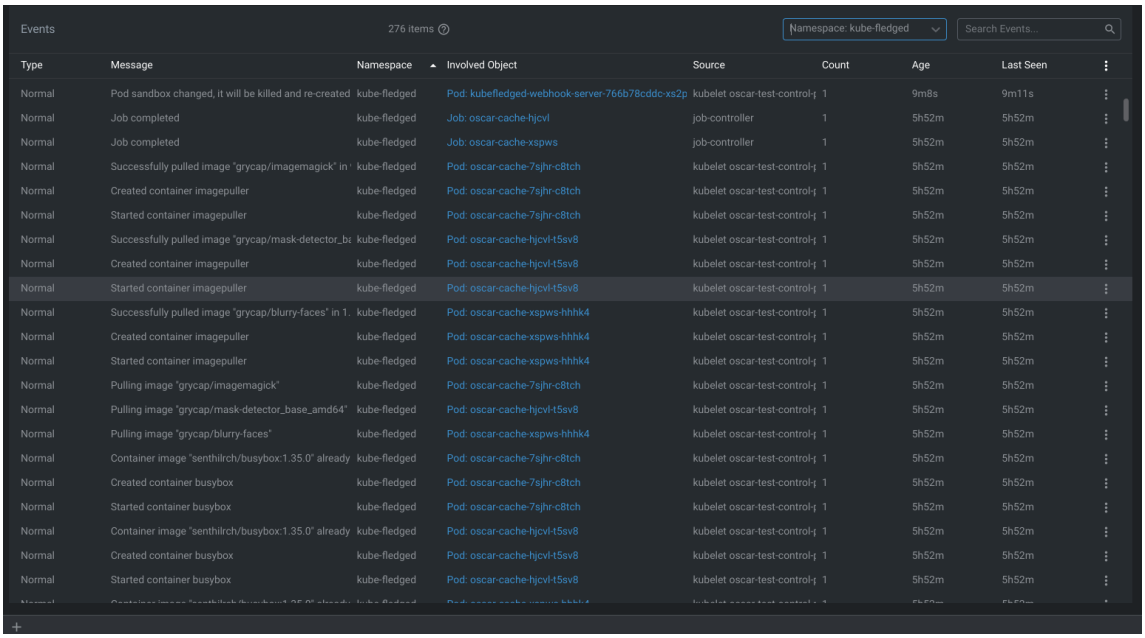


Figura 12 - Registro de eventos de OpenLens.

OpenLens permitirá llevar un rastreo de los eventos que se generen en OSCAR una vez ejecutados los servicios así como determinar el impacto de la caché en la ejecución

de dichos servicios. OpenLens permitirá obtener tiempos de descarga de imágenes y de ejecución de los servicios.

Otro aspecto importante del uso del OpenLens, es que permitirá diferenciar entre dos etapas en la ejecución de servicios: 1) tiempo de descarga de la imagen desde el repositorio y 2) el tiempo total de ejecución del servicio.

Capítulo 3 - Integración con OSCAR

En este capítulo se detalla cómo se ha integrado OSCAR con Kube Fledged y kube-image-keeper para habilitar el uso de caché de imágenes en las funciones sin servidor desplegadas en OSCAR. Se describirán los pasos realizados para configurar ambas soluciones de caché en el clúster de Kubernetes donde se ejecuta OSCAR. Además, se presentarán experimentos realizados para evaluar el impacto de la caché de imágenes en el rendimiento y costos de aplicaciones serverless en OSCAR. Los resultados obtenidos permitirán comprender los beneficios que aporta el uso de caché de imágenes en entornos sin servidor basados en contenedores.

3.1 Integración: Kube Fledged

Instalación/Desinstalación

El primer paso para integrar OSCAR y Kube Fledged consiste en realizar la respectiva instalación de la herramienta.

En este caso se cuenta con tres opciones para realizar la instalación de Kube Fledged y su respectiva integración con OSCAR.

Opción 1: Utilizar los paquetes de instalación *Helm*[2] que provee el desarrollador.
Opción 2: Descargar el código fuente, compilarlo e instalarlo.
Opción 3: Utilizar el archivo Makefile y las reglas definidas en él.

En este caso se optará por la opción número 3 por la sencillez del procedimiento. Este *Makefile* contiene las reglas necesarias, ya sea para instalar o desinstalar el sistema. Como se mencionó anteriormente, Kube Fledged funciona como una extensión para el clúster Kubernetes, por lo cual, su instalación o desinstalación no afecta al funcionamiento del clúster.

Se procede a su instalación mediante el siguiente comando:

"make deploy-using-yaml"

De este modo, se ejecutan los archivos YAML respectivos y se realizan automáticamente las configuraciones necesarias para la correcta instalación de la herramienta.

[2] *Helm* es un gestor de paquetes para Kubernetes. *Helm charts* hace referencia a los paquetes de las aplicaciones y sus respectivas dependencias

En este punto es importante hacer notar que Kube Fledged ofrece la posibilidad de configurar ciertos parámetros según sea necesario (Anexo 4)

Creación de la Caché

En el siguiente apartado, procederemos a estudiar los aspectos más importantes que es utilizada por Kube Fedged para crear la caché en los nodos.

En el ejemplo siguiente se observará un fragmento de código que define los campos específicos para crear una caché utilizando Kube Fledged.

```
---
apiVersion: kubefledged.io/v1alpha2
kind: ImageCache
metadata:
  name: oscar-cache
  namespace: kube-fledged
  labels:
      app: oscar
      kubefledged: imagecache
spec:
  cacheSpec:
  - images:
      - docker.io/grycap/imagemagick
```

Figura 13 - Creación de la caché en el clúster.

Del ejemplo anterior, se pueden destacar los siguientes puntos:

Sección Metadata: Mediante el selector *name* se define el nombre que se desea brindar a la caché. Además, el selector *namespace* permite especificar el *namespace* del cuál la caché formará parte. Este punto es importante ya que en Kube Fledged no se especifica ninguna ruta específica para guardar los datos, sino que la caché está conformada por pods.

Se debe tener en cuenta que al utilizar Kube Fledged, la caché estará compuesta por *pods* temporales que cumplirán el rol de almacenamiento para las imágenes indicadas en el manifiesto

Sección *spec*: En ella se especifica el selector *cacheSpec*, en la cual se especificarán las imágenes que se desea descargar a caché.

Las siguientes líneas deben ser agregadas al archivo YAML cuando se desee ubicar en caché una imagen para un nodo en específico (después del primer selector *images*).

Por último, si se requiere de una contraseña para acceder al registro de la imagen, se debe especificar el nombre del secreto a utilizar mediante el selector *name*.

```
- images:
  - docker.io/grycap/plant-classification-theano:latest
  nodeSelector:
    tier: <nombre del nodo>
- name: <secreto>
```

Figura 14 - Código para definir un nodo específico.

Importante mencionar que en caso de que se desee que una imagen se almacene en todos los nodos del clúster, se puede omitir la sección *nodeSelector*, la imagen será ubicada en la caché para todos los nodos del clúster.

Una vez definido el manifiesto para la creación de la caché, se procede a ejecutar el siguiente comando:

kubectl create -f kubefledged-imagecache.yaml

Otro comando importante es el que muestra la siguiente figura y que se anota a continuación, cuyo objetivo es mostrar el estado de la caché.

kubectl get imagecaches oscar-cache -n kube-fledged -o json

```
[oscar@fedora deploy]$ kubectl get imagecaches oscar-cache -n kube-fledged -o json
{
    "apiVersion": "kubefledged.io/v1alpha2",
    "kind": "ImageCache",
    "metadata": {
        "creationTimestamp": "2023-06-19T21:57:51Z",
        "generation": 3,
        "labels": {
            "app": "oscar",
            "kubefledged": "imagecache"
        },
        "name": "oscar-cache",
        "namespace": "kube-fledged",
        "resourceVersion": "38034",
        "uid": "07ff2caf-b3d7-4082-a0da-c3be9b324a39"
    },
    "spec": {
        "cacheSpec": [
            {
                "images": [
                    "grycap/imagemagick"
                ]
            }
        ]
    },
    "status": {
        "completionTime": "2023-06-19T21:58:23Z",
        "message": "All requested images pulled succesfully to respective nodes",
        "reason": "ImageCacheCreate",
        "startTime": "2023-06-19T21:57:51Z",
        "status": "Succeeded"
    }
}
[oscar@fedora deploy]$
```

Figura 15 - Creación de la caché en el clúster.

Uso de recursos

Una vez que Kube Fledged se encuentra en ejecución y se han descargado las imágenes a caché, cada imagen tendrá un pod distinto dentro del clúster, lo cual es un punto importante a tener en cuenta debido a que muchas imágenes en caché, significa un pod correspondiente para cada una de ellas, ocasionando una mayor demanda de almacenamiento en clúster.

Lo anterior se puede observar en la imagen mostrada abajo, donde solo se cuentan con tres imágenes en caché y cada una de ellas cuenta con un pod asignado (oscar-cache-7sjhr-c8tch, oscar-cache-hjcvl-t5sv8 y oscar-cache-xspws-hhhk4). Esto debe ser un punto a considerar cuando se planea tener en caché un número grande de imágenes. Ejemplo 1000 imágenes en caché = 1000 pods en ejecución en clúster.

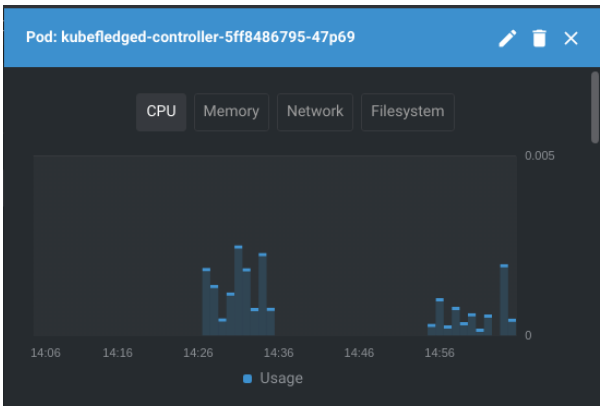
Normal	Successfully pulled image "grycap/imagemagick" in '	kube-fledged	Pod: oscar-cache-7sjhr-c8tch	kubelet oscar-test-control-[1
Normal	Created container imagepuller	kube-fledged	Pod: oscar-cache-7sjhr-c8tch	kubelet oscar-test-control-[1
Normal	Started container imagepuller	kube-fledged	Pod: oscar-cache-7sjhr-c8tch	kubelet oscar-test-control-[1
Normal	Successfully pulled image "grycap/mask-detector_ba	kube-fledged	Pod: oscar-cache-hjcvl-t5sv8	kubelet oscar-test-control-[1
Normal	Created container imagepuller	kube-fledged	Pod: oscar-cache-hjcvl-t5sv8	kubelet oscar-test-control-[1
Normal	Started container imagepuller	kube-fledged	Pod: oscar-cache-hjcvl-t5sv8	kubelet oscar-test-control-[1
Normal	Successfully pulled image "grycap/blurry-faces" in 1.	kube-fledged	Pod: oscar-cache-xspws-hhhk4	kubelet oscar-test-control-[1

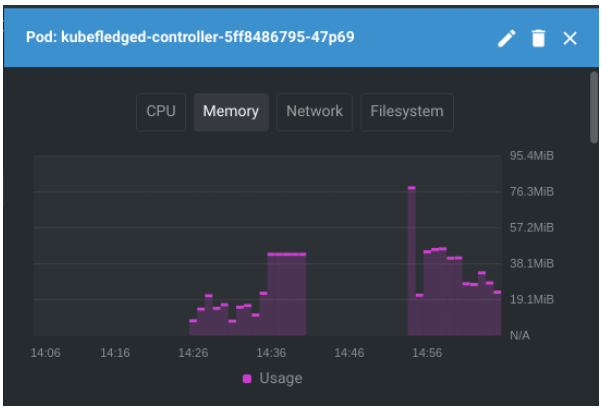
Figura 16 - Historial de ejecución Kube Fledged.

Se debe tener presente que el *Garbage Collector* de Kubernetes puede ocasionar la eliminación de las imágenes descargadas y almacenadas en la caché. Para estos casos, *Kube Fledged* cuenta con una rutina encargada de verificar si se han eliminado imágenes de la caché, y en caso de ser necesario, volver a descargarla.

En cuanto al uso de recursos, los podemos analizar mediante los siguientes gráficos obtenidos de la herramienta *OpenLens*, en los cuales se puede observar el uso que hace la herramienta de recursos como la CPU, la memoria, disco y sistema de archivos.

Controller:





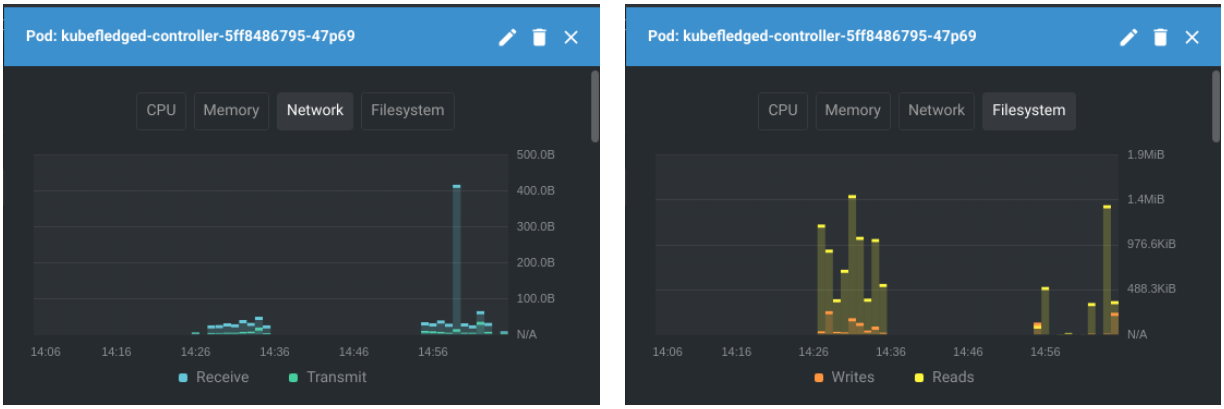


Figura 17 - Consumo de recursos del controlador de Kube Fledged.

Webhook:

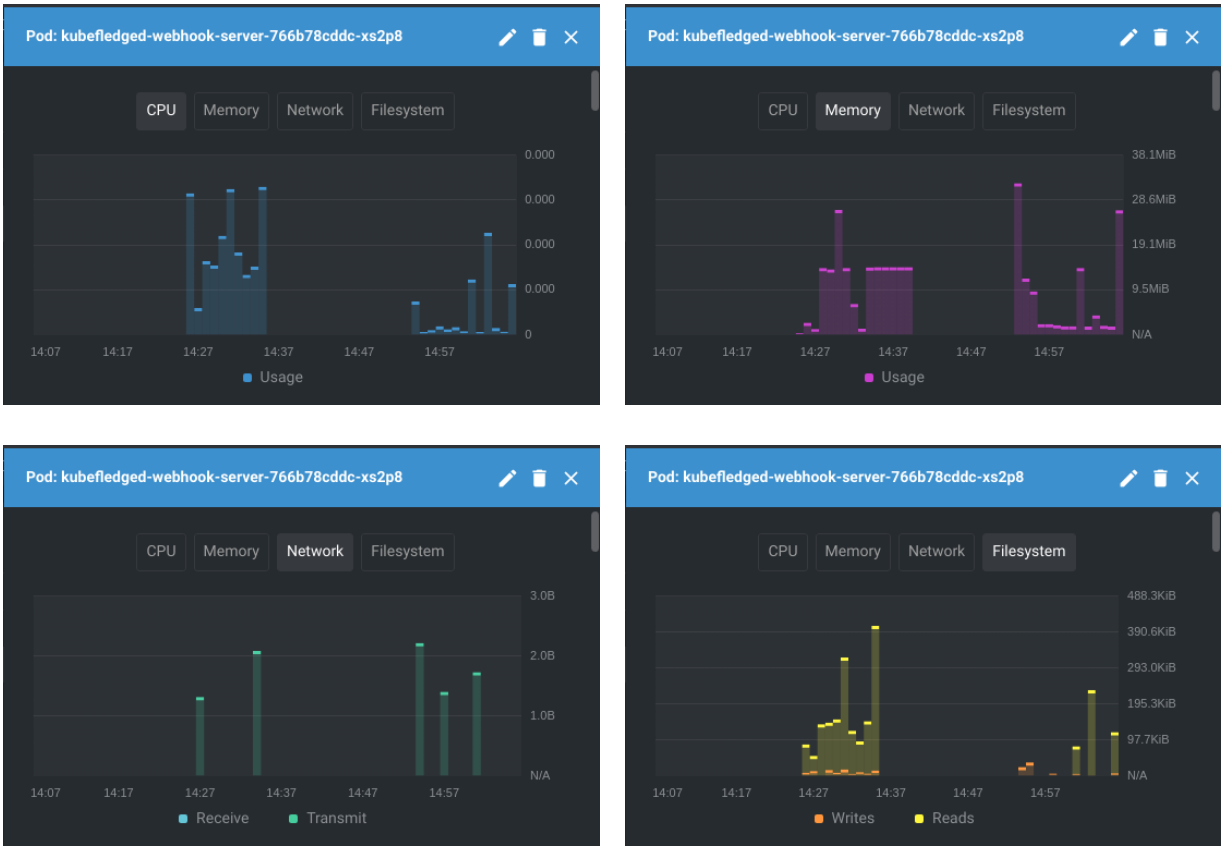


Figura 18 - Consumo de recursos del webhook server de Kube Fledged.

Las imágenes anteriores, permiten observar el consumo de recursos de los dos principales componentes de Kube Fledged. Como se puede notar, tienen un consumo de recursos mínimo en cada una de las diferentes categorías: CPU, Memoria, Red y Sistema de archivos.

3.2 Integración: kube-image-keeper (kuik)

Instalación/Desinstalación

La instalación de Kube-image-keeper se realiza de manera sencilla. El único requerimiento previo es contar con un *plugin Container Networking Interface* (CNI) que cuente con la funcionalidad *port-mapper*[3] habilitada por defecto.

Cumpliendo con lo anterior, se puede proceder a realizar la instalación de la herramienta prácticamente sin ninguna configuración adicional.

La instalación se realiza por medio de línea de comandos a través de un repositorio Helm o archivos YAML.

Al utilizar Helm se debe utilizar el siguiente comando [10]:

helm upgrade --install --create-namespace --namespace kuik-system \
kube-image-keeper kube-image-keeper --repo https://charts.enix.io/

Mediante archivos YAML se deben ejecutar los siguientes comandos [10]:

helm template --namespace kuik-system kube-image-keeper kube-image-keeper --repo https://charts.enix.io/ > /tmp/kuik.yaml
kubectl create namespace kuik-system
kubectl apply -f /tmp/kuik.yaml --namespace kuik-system

Creación de la Caché.

La creación de la caché se realiza de manera permanente, ya que se crea al momento de instalar la extensión.

Si se desea observar el estado de las imágenes en caché, se puede utilizar el siguiente comando [10]:

kubectl get cachedimages

A diferencia de Kube Fledged, kube-image-keeper no cuenta con un manifiesto para crear un recurso caché, sino que implementa un registro local de imágenes y un proxy, este último realiza las validaciones de imágenes, si la imagen está en caché, se

[3] Funcionalidad que permite asociar un puerto del contenedor con un puerto del host

distribuye la que se encuentra en la caché, de lo contrario, se descarga desde el repositorio de origen con su respectiva aa

Uso de recursos

Los requerimientos de kuik en cuanto a su ejecución son mínimas. A continuación, se muestran algunas métricas obtenidas desde OpenLens que permite visualizar el rendimiento de cada uno de sus componentes:

Controller: encargado de monitorear pods y la caché y crear recursos

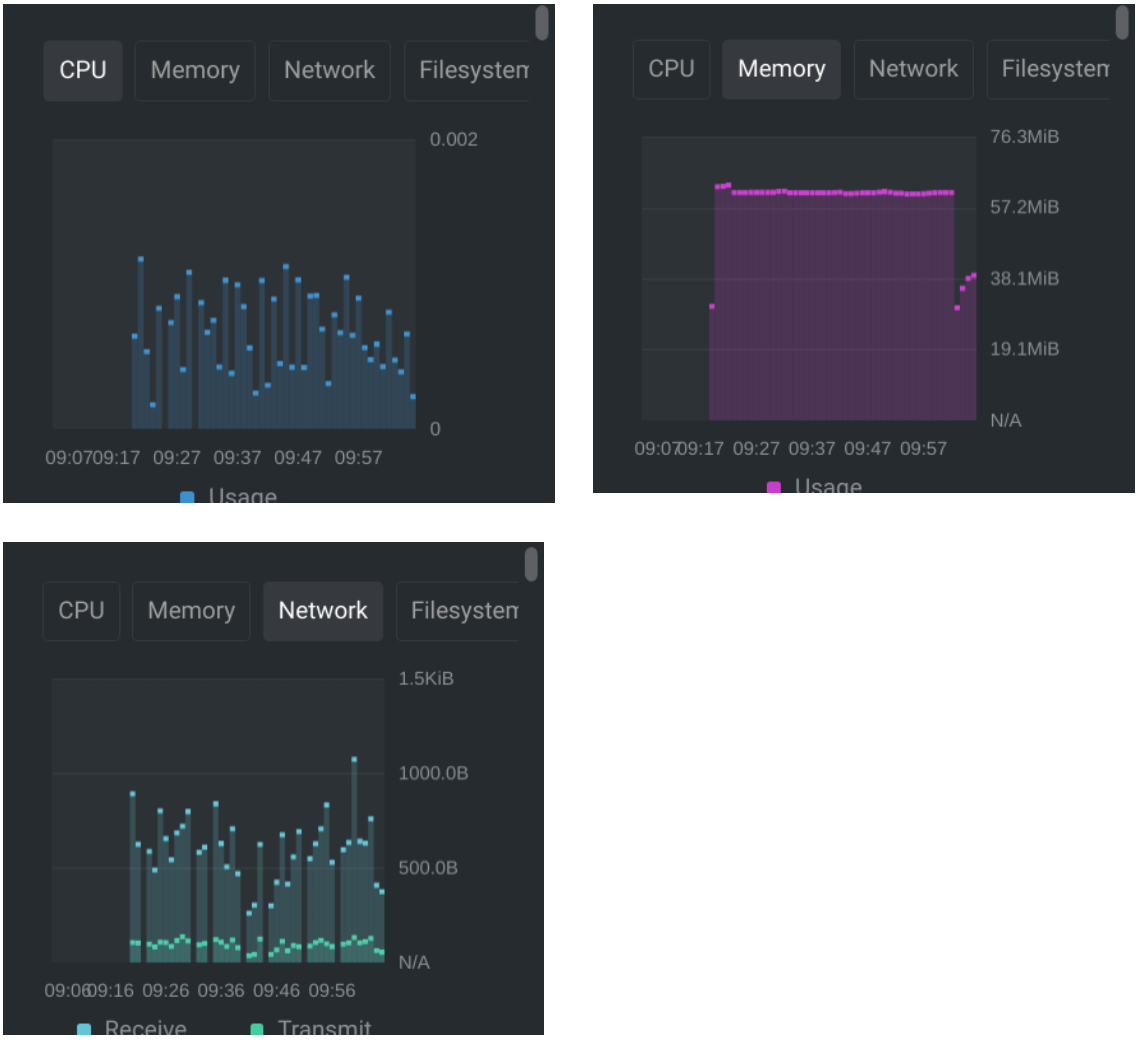


Figura 19 - Consumo de recursos del controller de kuik.

Proxy: Recibe peticiones y valida si es necesaria su descarga o redirigir la petición a la caché

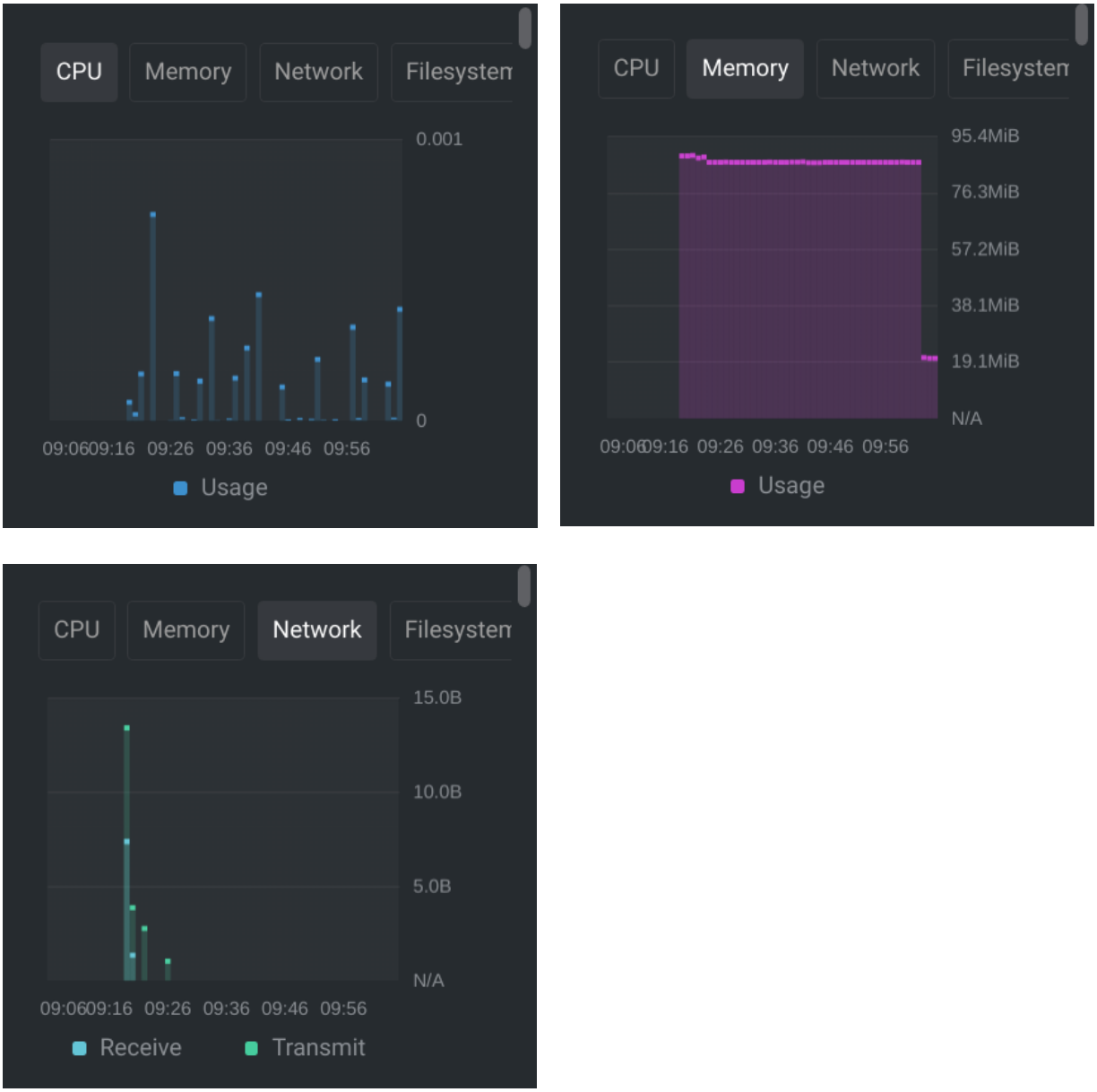


Figura 20 - Consumo de recursos del proxy de kuik.

Como se puede notar de los gráficos anteriores, el consumo de recursos por parte de la extensión es poco, por lo que tiene una poca afectación en el clúster.

Webhook: Encargado de modificar los manifiestos de los pods y actualizar su registro

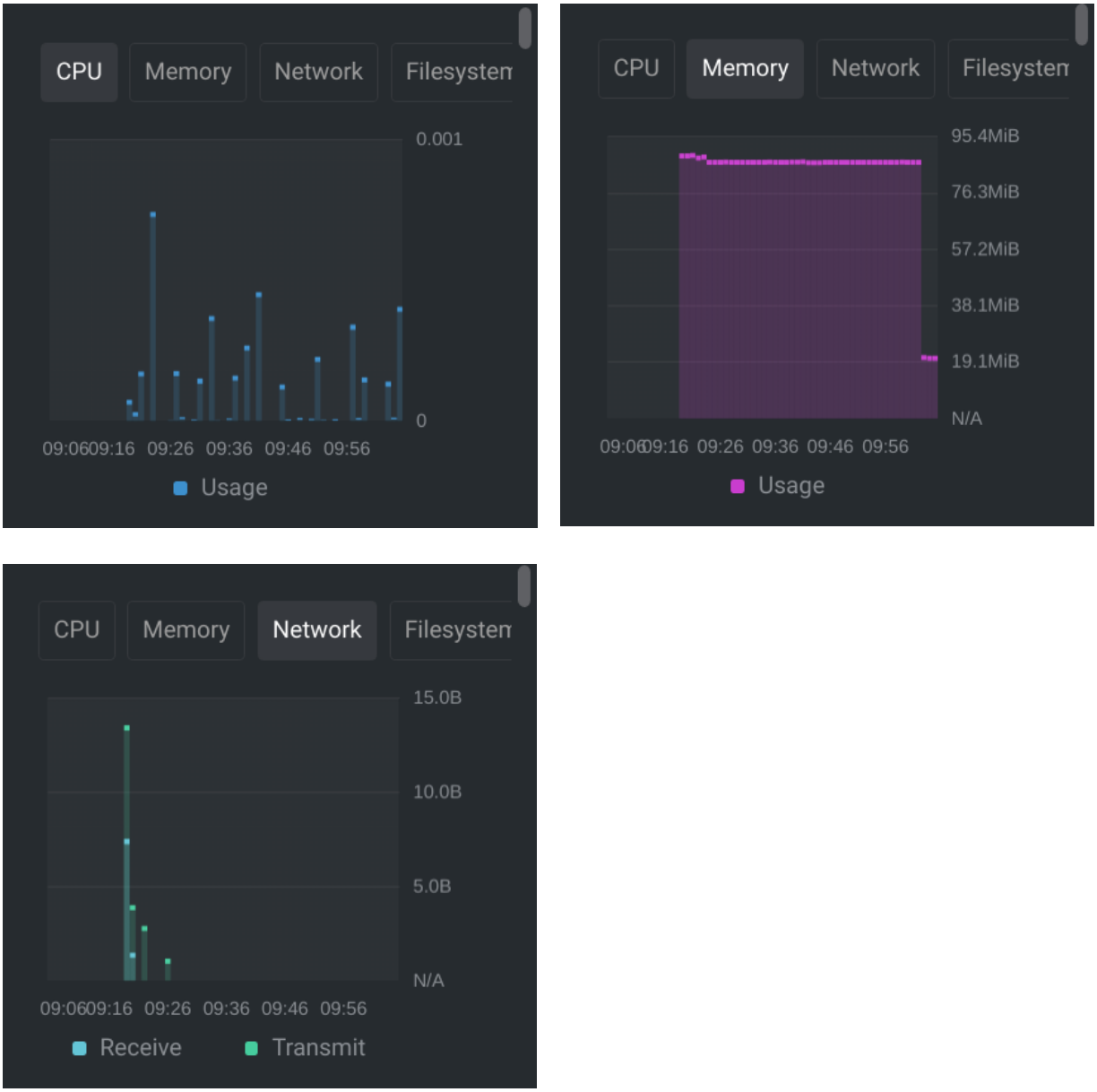


Figura 21 - Consumo de recursos del webhook kuik.

3.3 Tabla comparativa: Kube Fledged vs kube-image-keeper

A continuación se presenta una tabla comparativa de ambas herramientas, las cuáles han sido elegidas ya que son las más se adaptan al requerimiento de cacheo de imágenes y que no ofrecen soluciones dependientes de redes de alta velocidad o sistemas de distribución de imágenes.

Criterios	Kube Image Keeper (Kuik)	Kube Fledged
Declaración explícita de imágenes a descargar	- No posee manifiesto explícito	- Cuenta con manifiesto. (cuáles imágenes y en cuáles nodos deben descargarse)
Instalación mediante Helm	- Si	- Si
Detección automática de imágenes	- Automáticamente, detecta imágenes presentes en el nodo y las ubica en caché	- No detecta imágenes automáticas. La versión 0.11, incorporará esta funcionalidad
Definición de la expiración de imágenes	- Permite definir una fecha de expiración para cada una de las imágenes.	- No provee dicho parámetro
Monitoreo	- Provee una lista de métricas y *dashboards* compatibles con Grafana	- No provee una manera *out-of-the-box* para monitorear la herramienta
Tipo de cache	- Persistente	- No persistente. Utiliza pods temporales en el clúster

3.4 Configuración de servicios en OSCAR

En OSCAR, un servicio se puede definir como una representación de una función *serverless*, además, el servicio involucra la especificación de la cantidad de CPU, memoria requeridos así como el sistema de almacenamiento a utilizar (Minio, S3, Onedata).

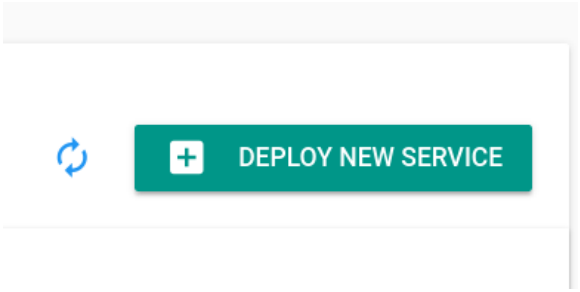
Al utilizar Minio y subir archivos en la plataforma, se genera un evento en OSCAR que provoca la creación de un pod en la plataforma. De este modo, al subir N archivos en Minio se generarán N eventos que crearán N pods, los cuales que se ejecutan de forma eficiente sobre el clúster y cuyo número de nodos puede crecer y decrecer en función de las necesidades de cómputo.

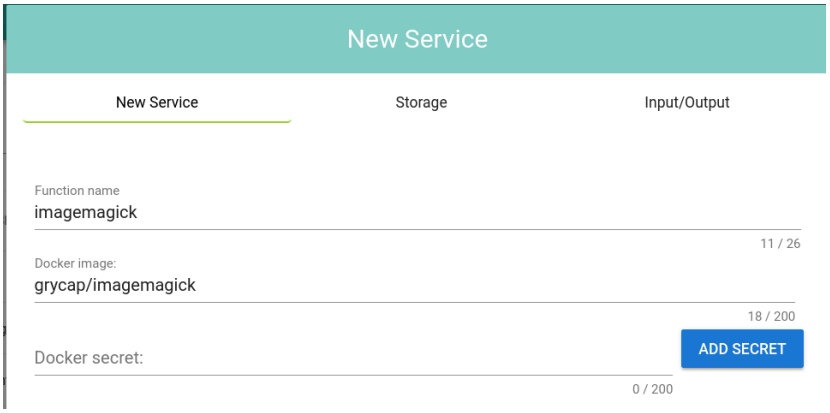
Antes de analizar el comportamiento de las herramientas, se dará un paso a paso de cómo crear un servicio de OSCAR para ejecutar los trabajos necesarios.

Para la configuración de servicios en OSCAR se cuentan con varias opciones. 1) Creación por medio del *Function Definition Language*[4], 2) utilizando la línea de comandos de OSCAR y 3) por medio de la interfaz gráfica de OSCAR. En esta ocasión, se utilizará la interfaz de usuario para la definición de servicios.

Paso 1:

Al hacer clic en el botón de *Deploy New Service*, se despliega la ventana *New Service*. En dicha ventana se define el nombre de la imagen en el campo *Function Name* así como la imagen necesaria en el espacio *Docker Image*.

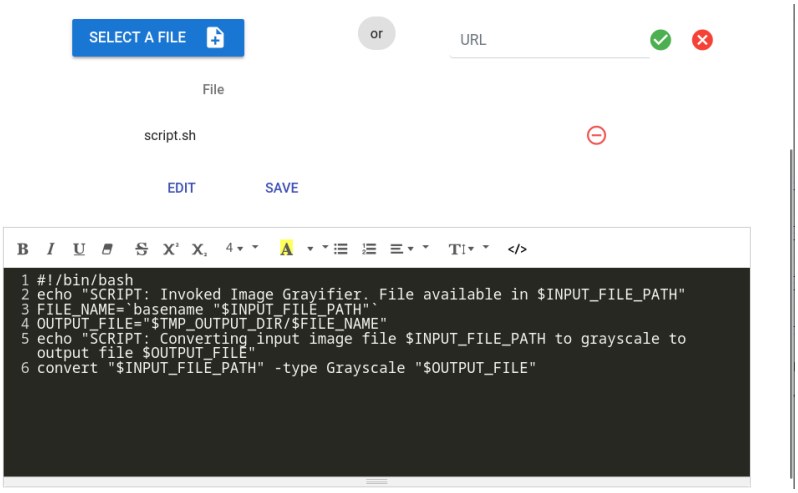




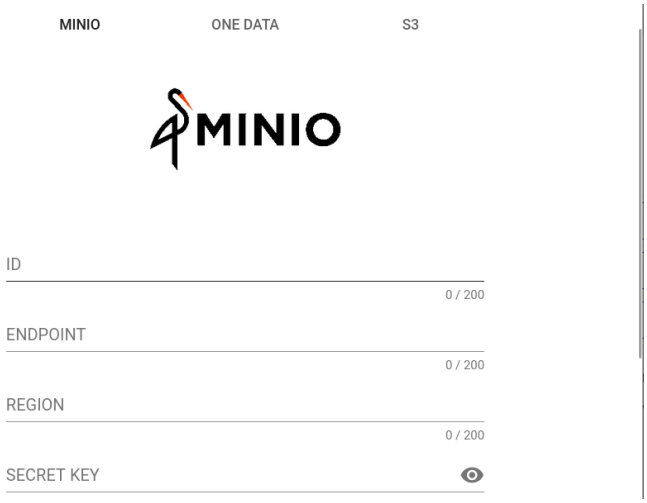
Paso 2: En la misma ventana, se deberá ingresar un script el cual invocará los comandos necesarios para la ejecución de la función. Este *script* corresponde a un archivo y si se requiere, se puede editar el *script*.

[4] *Function Definition Language* (FDL) permite definir una especificación del servicio a crear en la plataforma OSCAR. Ejemplo en el Anexo

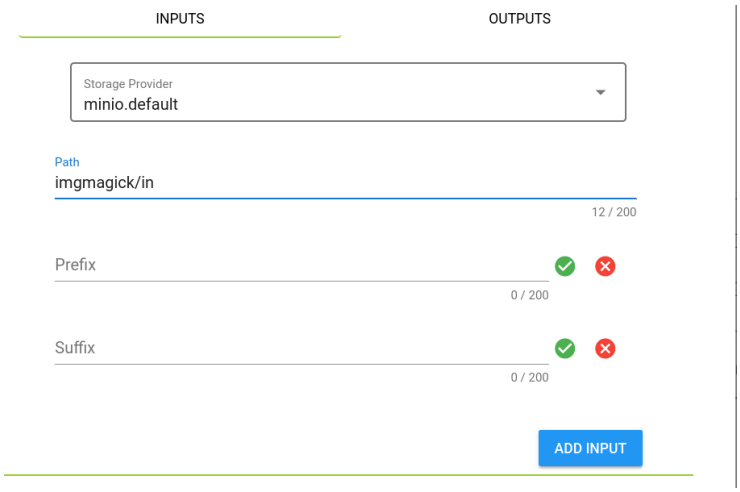
Este script permitirá la ejecución del código dentro del contenedor. El ejemplo utilizado corresponde al script utilizado para convertir imágenes a escala de grises mediante la librería *imagemagick* y el comando *convert*.

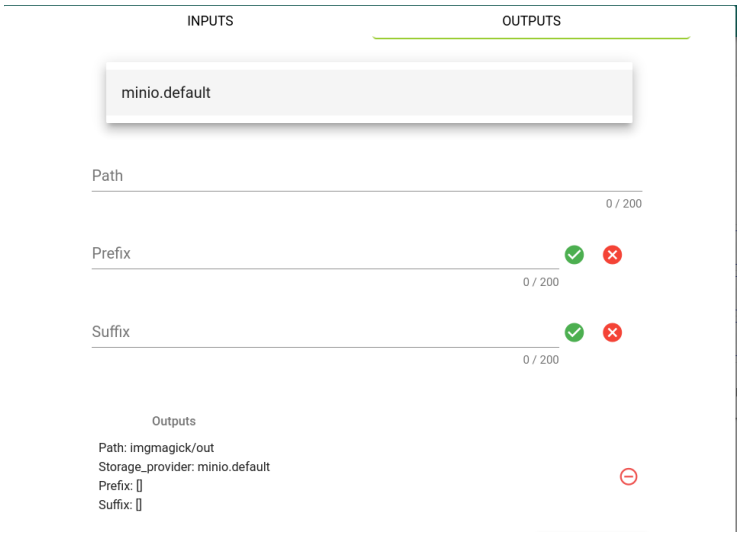


Si se va a utilizar un sistema de almacenamiento específico, se puede configurar Minio, Onedata o S3.

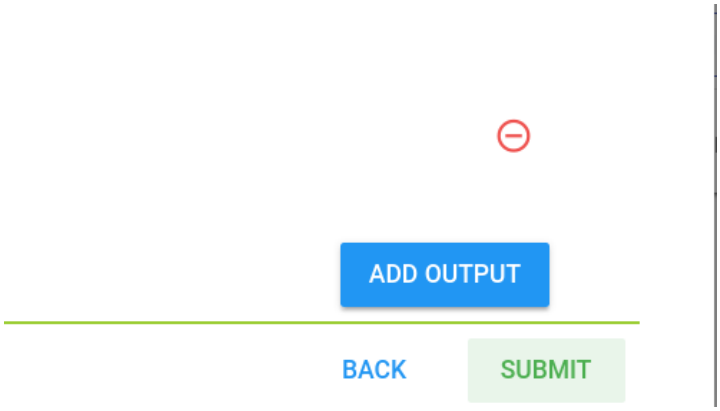


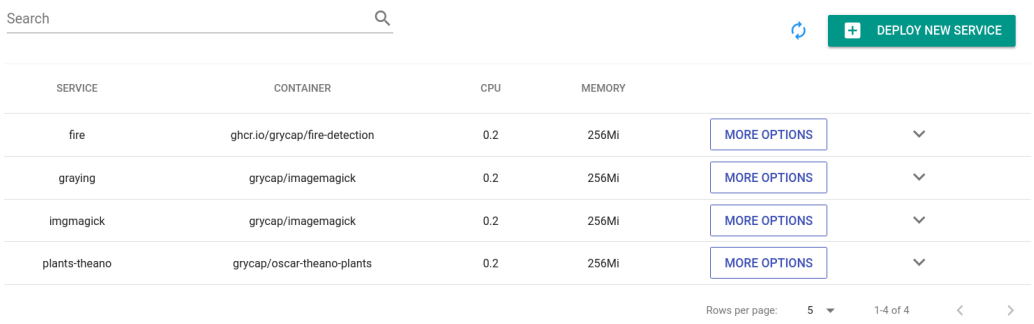
Finalmente, se define la ubicación de los archivos que se utilizarán para la ejecución de la función. OSCAR brinda una conexión *out-of-the-box* con Minio por lo que se utilizará esa opción. Se selecciona la opción minio.default en el campo *Storage Provider*.



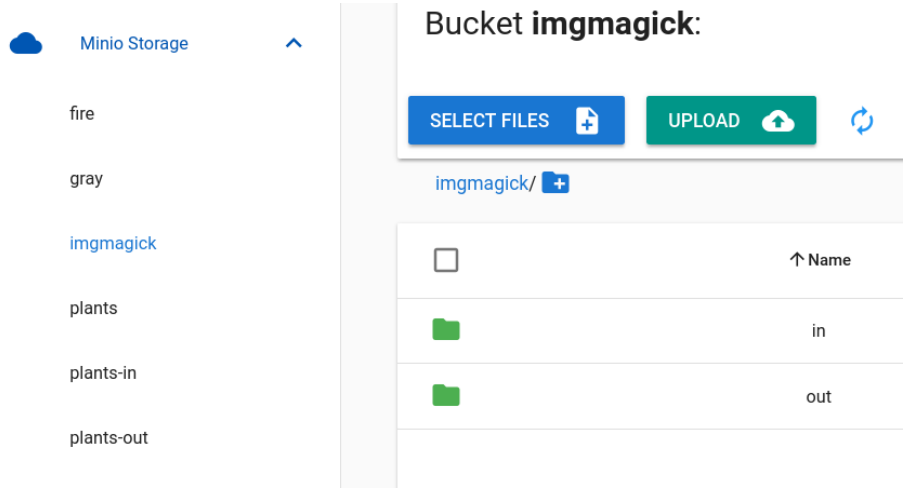


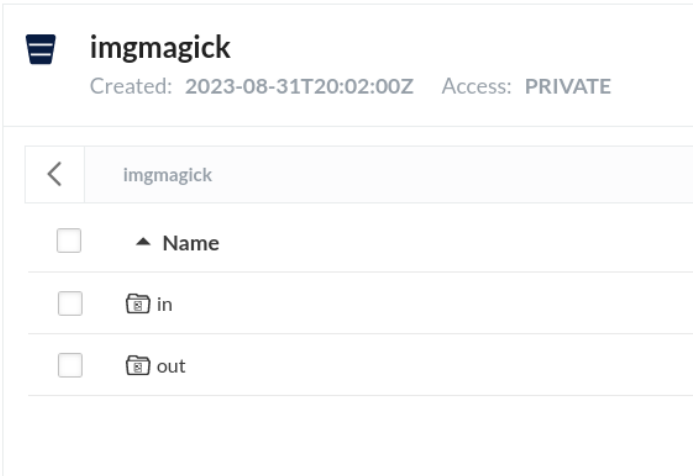
Finalmente, hacer clic en el botón *Submit* y la plataforma procederá a la creación del servicio.





En las imágenes siguientes, se puede corroborar la creación de los *buckets*, así como los directorios definidos anteriormente, tanto en desde OSCAR como en Minio.





Capítulo 4 - Pruebas

En este capítulo se observarán los resultados obtenidos al realizar las pruebas con las herramientas Kube Fledged y kuik. Se observarán tiempos de descarga de cada ejecución así como los tiempos totales de ejecución.

4.1 Entorno de ejecución

Para realizar las pruebas se utilizaron 3 máquinas virtuales, 1 máquina libre de cualquier herramienta de optimización y que servirá para tomar información que se utilizará como para comparar contra los datos obtenidos desde las máquinas de Kube Fledged y kuik. Las máquinas cuentan con las siguientes características:

- Procesador: AMD Ryzen™ 7 5800H
- Memoria RAM: 8mb
- Sistema Operativo: Fedora® 38
- Programa de virtualización: Oracle® VM VirtualBox

4.2 Ejecución de las pruebas

Para realizar el experimento se ha tomado como base la imagen *grycap/imagemagick [45]*. La imagen instala la librería *imagemagick*, cuya funcionalidad es convertir una imagen (recibida como entrada), a otra en escala de grises. Además de la instalación, se configura la variable de ambiente *PATH* para habilitar los comandos de la librería, a nivel global dentro del contenedor. Se ha elegido esta imagen porque permite una ejecución rápida y a su vez observar el comportamiento de las herramientas ante una carga de trabajo considerable.

Para la evaluación, se ejecutan en OSCAR 75 trabajos que utilizan la función *imagemagick* para estudiar el efecto de utilizar una caché junto a OSCAR.

Como se mencionó anteriormente, al subir un archivo en Minio se crea un evento que crea un pod automáticamente, de esta manera, el sistema crea los servicios en OSCAR automáticamente y a su vez, los pods necesarios para ejecutar la carga de trabajo mencionada, en este caso en específico, un pod por servicio.

Importante recalcar que para obtener la información se utilizó el comando mostrado abajo junto a la herramienta OpenLens, la cual, por medio de su interfaz de usuario permite observar los eventos que ocurren en el clúster Kubernetes mientras se

ejecutan los servicios de OSCAR. La imagen siguiente muestra la cómo se observa un evento desde la interfaz gráfica de OpenLens:

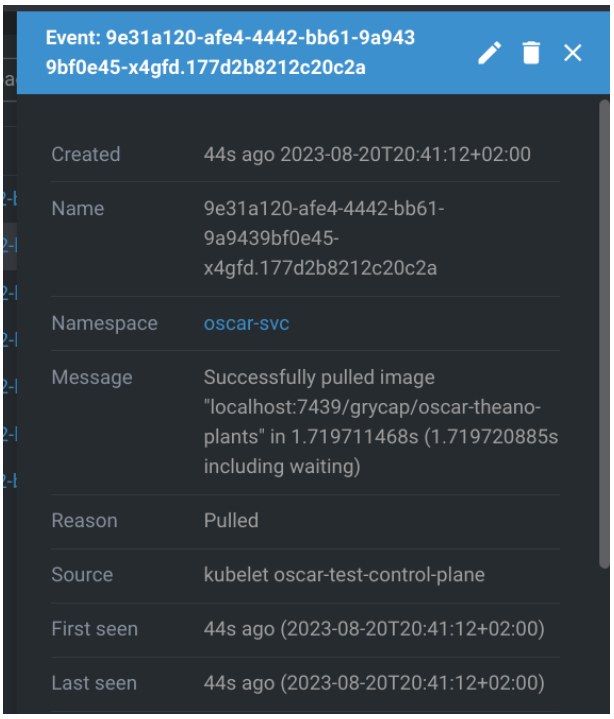


Figura 22 - Eventos desde la interfaz gráfica de OpenLens.

kubectl get events --namespace oscar-svc | grep -E 'Successfully pulled image' > resultados.txt

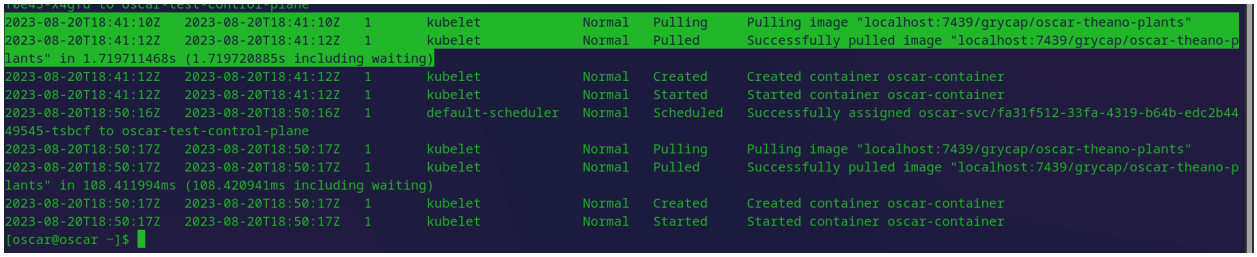


Figura 23 - Eventos obtenidos por línea de comandos.

El comando anterior brinda la información requerida como el tiempo de descarga y los tiempos totales de ejecución. En este caso, solo interesan los eventos que contengan la línea *Successfully pulled image*, lo que es suficiente para obtener los tiempos requeridos para realizar la comparación entre las herramientas.

Las siguientes son las razones que justifican el número de trabajos que se eligió:

- Un tamaño de muestra grande permite una mejor representación de la población objetivo y reduce el error de muestreo. Con una muestra pequeña de pruebas, los resultados podrían verse sesgados.
- Realizar pruebas en una gran cantidad de trabajos permite analizar una mayor variedad de casos y escenarios. En este caso, 75 trabajos permiten probar la solución propuesta de manera más completa.
- Un tamaño de muestra grande fortalece la significancia estadística de los resultados. Es más difícil que una cantidad considerable de trabajos muestren resultados por casualidad en comparación con una muestra pequeña.
- Restricciones de recursos.

4.3 Hallazgos

Una vez ejecutadas las pruebas y extraída la información, se obtienen los datos mostrados en las imágenes siguientes. En ellas se pueden observar los tiempos de descarga de imágenes y tiempos totales.

Dado que se está estudiando el impacto de una caché en sistemas *serverless*, es importante contar con este dato debido a que es el que permitirá determinar el impacto de una caché funcionando junto a OSCAR.

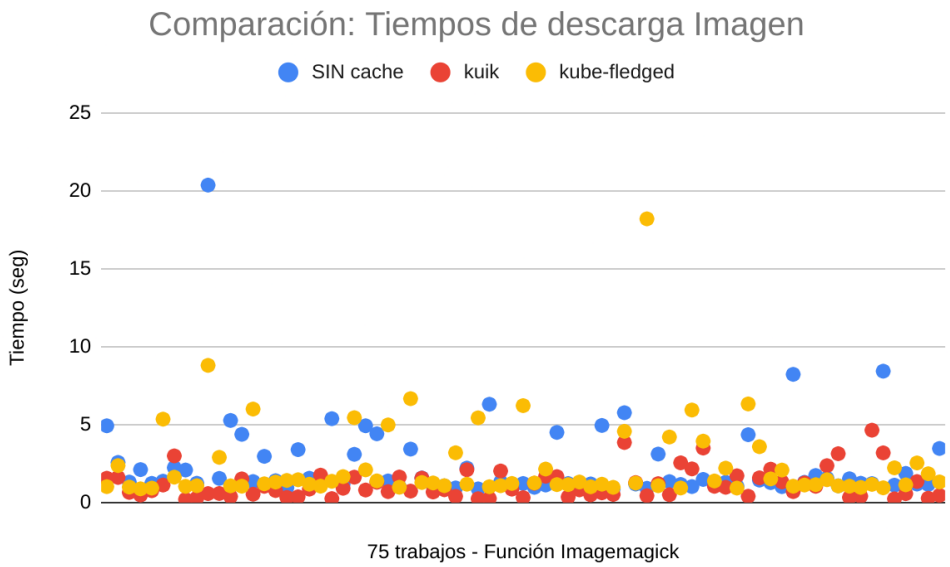


Figura 24 - Tiempos de descarga de imágenes.

De la imagen anterior, se puede notar que kuik ofrece una disminución de tiempos notable con respecto a Kube Fledged. Recordando el funcionamiento de kuik, esta

herramienta funciona con un proxy que valida si la imagen se encuentra en un registro local o tiene que ser descargada del repositorio. Al detectar que la imagen ya se encuentra en caché, se procede a utilizar dicha versión en lugar de descargarla del repositorio.

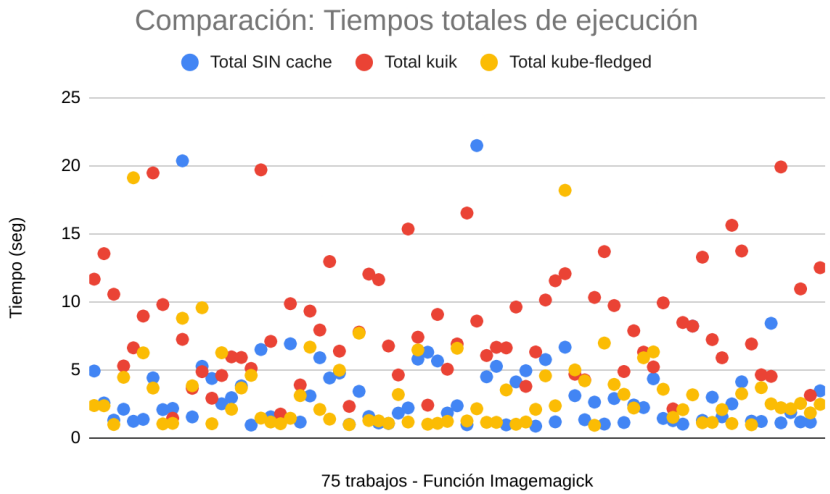


Figura 25 - Tiempos totales de ejecución.

La imagen anterior muestra la ejecución de una función *serverless* en su totalidad, desde la descarga de la imagen o acceso a la caché hasta su ejecución y su consecuente finalización.

Interesante notar la diferencia de tiempos que se genera con kuik, ya que el tiempo de descarga de imágenes disminuye notablemente, pero la ejecución total del servicio, es mayor (desde su inicio hasta su finalización). Lo anterior puede deberse a que se ingresan una cantidad elevada de trabajos en la cola de procesamiento, lo cual supera la capacidad de ejecución del sistema y genera un cuello de botella en OSCAR, aumentando los tiempos de espera. Además, al ejecutarse en máquina virtual puede darse una afectación en el rendimiento ya que se cuentan con recursos limitados y pueden existir diferencias si se ejecutan en entornos con una mayor capacidad.

En la siguiente tabla se puede observar el promedio de los tiempos con las herramientas y sin ellas. De manera gráfica, se puede observar la imagen abajo mostrada y que permite observar

	Tiempo SIN caché (seg)	**Tiempo Total SIN caché (seg)**	**Tiempo kuik (seg)**	**Tiempo Total kuik (seg)**	**Tiempo kube-fledged (seg)**	**Tiempo Total kube-fledged (seg)**
Tiempos	2.46	3.41	1.17	8.24	2.36	3.36

promedio						

Figura 26 - Tabla de tiempos promedio de descarga y de ejecución.

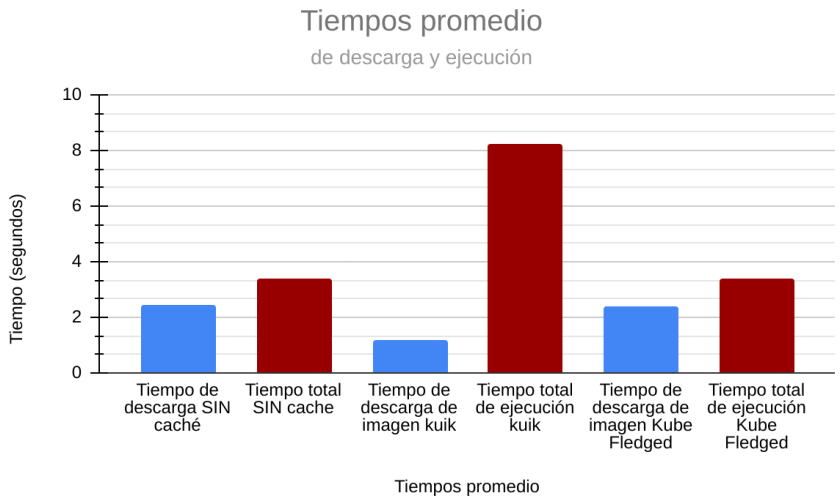


Figura 27 - Tiempos promedio de descarga y de ejecución.

En cuanto a kube-fledged, muestra una ligera mejoría con respecto a la ejecución sin caché. Esta herramienta mantiene un porcentaje de descarga y ejecución lineal, es decir, no hay cuellos de botella que afecten la ejecución del servicio.

Capítulo 5 - Conclusiones

En el siguiente apartado se mencionan algunas de las conclusiones a las que se ha llegado después del estudio realizado.

- Al utilizar sistemas serverless, se deben agregar sistemas que nos permitan medir el rendimiento de la aplicación, de modo que se puedan determinar y tomar las acciones de mejora más oportunas.

- Si se desea emplear un proveedor en la nube, se debe analizar las herramientas y opciones que ofrece dicho proveedor para remediar los desafíos que presenta la computación serverless.

- Entender y tener claro el objetivo de por qué se desea usar un método de caché en el clúster.

- Cualquiera de los enfoques que se sigan para atenuar el arranque en frío, se debe tener presente que conlleva ventajas y desventajas.

- Al escoger la herramienta de caché, se debe tener claro el tipo de caché que se desea, ya que se observó en las herramientas anteriores, kuik ofrece una caché persistente, mientras Kube Fledged no, dado que la caché son pod creados y ejecutados en clúster.

Capítulo 6 - Trabajos Futuros

Como se ha visto, existen varios involucrados en el mundo tecnológico que están tratando de brindar distintas soluciones para resolver el problema del arranque en frío, ya sea desde proveedores de nubes públicas hasta iniciativas de código abierto.

Para este trabajo en específico se puede pensar en el despliegue de las herramientas de cacheo en una infraestructura con una mayor cantidad de nodos, de modo que permita estudiar el comportamiento de las herramientas en entornos más complejos. Además, utilizar diversos tamaños de imágenes, estudiar la factibilidad de combinar estrategias como el cacheo, contenedores calientes o tibios. Inclusive se puede valorar el desarrollo interno de alguna herramienta interna de cacheo adaptada a los requerimientos específicos.

Anexos

Anexo 1: Relación con los objetivos de desarrollo sostenible de la ONU

En el siguiente anexo se presentan los Objetivos de Desarrollo Sostenible (ODS) de la Organización de las Naciones Unidas (ONU) que guardan relación directa con el tema "Estrategias de Mitigación de Arranque en Frío en Plataformas *Serverless On-Premises*".

Objetivo 9: Industria, innovación e infraestructura.
Este objetivo se relaciona con el uso de estrategias de mitigación de arranque en frío en plataformas serverless, ya que implica el desarrollo y la implementación de soluciones tecnológicas innovadoras para mejorar la infraestructura y la eficiencia en la entrega de servicios.

Objetivo 11: Ciudades y comunidades sostenibles.
El uso de plataformas serverless con estrategias de mitigación de arranque en frío puede contribuir a la creación de ciudades y comunidades sostenibles, al mejorar el rendimiento de las aplicaciones y reducir el consumo de recursos, lo que a su vez puede promover una mayor eficiencia energética y una gestión más eficiente de la infraestructura tecnológica.

Objetivo 13: Acción por el clima.
Al implementar estrategias de mitigación de arranque en frío en plataformas *serverless*, se puede reducir el consumo innecesario de recursos y, por lo tanto, contribuir a la mitigación del impacto ambiental y la huella de carbono asociada con la ejecución de aplicaciones en la nube.

Anexo 2: Acrónimos

API - Interfaz de Programación de Aplicaciones.

AWS - Amazon Web Services.

CLI - Interfaz de Línea de Comandos

CNI - Container Networking Interface

CPU - Unidad Central de Procesamiento

CRD - CustomResourceDefinition

GB - Gigabyte

GCP - Google Cloud Platform

IoT - Internet de las Cosas

MIT - Instituto Tecnológico de Massachusetts

OCI - Iniciativa de Contenedores Abiertos

ONU - Organización de Naciones Unidas

P2P - Peer-to-peer

TB - Terabyte.

UI - Interfaz de Usuario.

YAML - YAML Ain't Markup Language.

Anexo 3: Kube Fledged - Manifiesto para crear caché

Kube-fledged requiere de un manifiesto YAML para crear una caché. A continuación se presenta el manifiesto completo con cada una de sus secciones.

```yaml
---
apiVersion: kubefledged.io/v1alpha2
kind: ImageCache
metadata:
  name: imagecache1
  namespace: kube-fledged
  labels:
    app: kubefledged
    kubefledged: imagecache
spec:
  cacheSpec:
  - images:
    - ghcr.io/jitesoft/nginx:1.23.1
  - images:
    - us.gcr.io/k8s-artifacts-prod/cassandra:v7
    - us.gcr.io/k8s-artifacts-prod/etcd:3.5.4-0
    nodeSelector:
      tier: backend
  imagePullSecrets:
  - name: myregistrykey
```

Figura 28 - Consumo de recursos del controlador de Kube Fledged.

A continuación, se brinda una explicación de cada una de sus partes:

- **apiVersion**: Especifica la versión del API del objeto Kubernetes. Para el caso de *kube-fledged* se utiliza la versión kubefledged.io/v1alpha2.
- **kind**: Especifica el tipo de objeto Kubernetes a utilizar. En este caso se utiliza un objeto *ImageCache*.
- **metadata**: Se definen los meta datos para el objeto *ImageCache*.
- **name**: Se define el nombre que se desea para el objeto *ImageCache*.
- **namespace**: Espacio de nombre en el cuál se desea crear el recurso caché.
- **labels**: Etiquetas que se desea aplicar al objeto *ImageCache*.
- **spec**: Sección del manifiesto en el que se procede a definir el objeto *ImageCache*.

- **cacheSpec**: En esta sección se especifican las imágenes a descargar y en cuáles nodos se deben descargar. Esta área se compone de:
    - Primera etiqueta *images*: en esta etiqueta se listan las imagenes que serán descargadas en todos los nodos del clúster
    - Segunda etiqueta *images*: Esta etiqueta se indican las imágenes a descargas en nodos específicos.
- **nodeSelector**: En esta sección se definen los nodos en los cuáles serán descargadas las imágenes definidas en la etiqueta anterior
- **imagePullSecrets**: Sección en la que se definen las contraseñas/secretos, en caso de ser necesarios para la descarga de las imágenes. Esto se realiza mediante el la clave-valor: name: <secreto_a_utilizar>

Anexo 4: Parámetros configurables para Kube Fledged

Los siguientes corresponden a la lista de parámetros configurables en la herramienta Kube Fledged.

Estos parámetros se pueden añadir al momento de instalar la herramienta, en el archivo YAML llamado *kubefledged-deployment-controller.yaml* o en el archivo values que es utilizado en el paquete Helm.

- **--cri-socket-path**: ruta al socket cri en el nodo, (predeterminado: /var/run/docker.sock, /run/containerd/containerd.sock, /var/run/crio/crio.sock)
- **--image-cache-refresh-frequency**: Define cada cuánto tiempo se actualizan las imágenes en caché. Establecer esta bandera en "0s" deshabilitará la actualización. Valor predeterminado "15m"
- **--image-delete-job-host-network**: Para eliminar imágenes se ejecuta un pod con dicha tarea. Esta bandera define si dicho pod se debe ejecutar utilizando la red del nodo o no (*'HostNetwork: true*' o *'HostNetwork: false*').
- **--image-pull-deadline-duration**: Duración máxima permitida para extraer una imagen. Después de esta duración, se considera que la extracción de imágenes ha fallado. valor predeterminado "5m"
- **--image-pull-policy**: Política a utilizar para descargar imágenes y actualizar el caché. Los valores posibles son 'IfNotPresent' y 'Always'. El valor predeterminado es 'IfNotPresent'. La imagen sin o con etiqueta ":latest" siempre se extrae.
- **--job-priority-class-name**: priorityClassName de los trabajos creados por kubefledged-controller.
- **--job-retention-policy**: Determina si los trabajos creados por kubefledged-controller se eliminarán o retendrán (para depuración) después de finalizar. Los valores posibles son 'delete' y 'retain'. El valor predeterminado es 'delete'.
- **--service-account-name**: cuenta de servicio utilizada en los jobs creados para descargar o eliminar imágenes. Es un campo opcional. Si no se especifica, se utiliza la cuenta de servicio predeterminada del espacio de nombres.
- **--stderrthreshold**: Nivel de registro de errores. El valor por defecto INFO

Anexo 5: Ejemplo de una especificación Function Definition Language

A continuación se muestra un ejemplo de una especificación FDL, de modo que permite la definición de un servicio en OSCAR.

Como se observa en el ejemplo siguiente, en dicha especificación se definen, al igual que en la interfaz de usuario, el nombre del servicio, cantidad de memoria y CPU a utilizar, imagen Docker a descargar, nombre del archivo script, las rutas en Minio a utilizar para tomar archivos de entrada y guardar los resultados y parámetros de escalado.

```
functions:
  oscar:
  - oscar-cluster:
     name: body-pose-detection-async
     memory: 2Gi
     cpu: '1.0'
     image: deephdc/deep-oc-posenet-tf
     script: script.sh
     environment:
        Variables:
          INPUT_TYPE: json
     expose:
      min_scale: 1
      max_scale: 10
      port: 5000
      cpu_threshold: 20
     input:
     - storage_provider: Minio.default
       path: body-pose-detection-async/input
     output:
     - storage_provider: Minio.default
       path: body-pose-detection-async/output
```

Figura 29 - FDL para crear un servicio[5]

[5] Tomado de https://docs.oscar.grycap.net/invoking/

Anexo 6: Capas de la imagen Docker *imagemagick*

La siguiente imagen muestra las capas que conforman la imagen Docker *imagemagick*.

Figura 30 - Capas de la imagen Docker imagemagick

Referencias

[1] Şamdan, E. (2021, Enero 23). *Solving Cold Starts in Serverless Architecture*. Built In. Accedido en Junio 24, 2023, desde https://builtin.com/software-engineering-perspectives/cold-starts-challenge-serverless-architecture

[2] Amaral, C. (2023, Marzo 1). *How to deal with serverless cold start: tips and solutions*. bohr.io. Accedido en Junio 24, 2023, desde https://blog.bohr.io/cold-start-in-serverless-the-overcome-problem-that-still-generates-prejudice

[3] Senthil Raja Chermapandian. Kube-fledged: Cache Container Images in Kubernetes. Accedido desde https://devpress.csdn.net/k8s/62ebf98519c509286f415f1b.html

[4] *Use Image streaming to pull container images | Google Kubernetes Engine (GKE)*. (n.d.). Google Cloud. Accedido en Junio 24, 2023, desde https://cloud.google.com/kubernetes-engine/docs/how-to/image-streaming

[5] P. Vahidinia, B. Farahani and F. S. Aliee, "Cold Start in Serverless Computing: Current Trends and Mitigation Strategies," *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, Barcelona, España, 2020, pp. 1-7, doi: 10.1109/COINS49042.2020.9191377.

[6] *oscar/README.md at master · grycap/oscar · GitHub*. (n.d.). GitHub. Accedido en Junio 17, 2023, desde https://github.com/grycap/oscar/blob/master/README.md

[7] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: towards high-performance serverless computing. In Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18). USENIX Association, USA, 923–935.

[8] *senthilrch/kube-fledged: A kubernetes operator for creating and managing a cache of container images directly on the cluster worker nodes, so application pods start almost instantly*. (n.d.). GitHub. Accedido en Junio 17, 2023, desde https://github.com/senthilrch/kube-fledged

[9] Raja, S. (2021, Seteimbre 27). *Kube-fledged: Cache Container Images in Kubernetes | by Senthil Raja Chermapandian*. ITNEXT. Accedido en Juio 18, 2023, desde https://itnext.io/kube-fledged-cache-container-images-in-kubernetes-7880a00bab91

[10] *Your k8s images, in your cache, close to your cluster(s)*. (2023, Enero 25). Enix.io. Accedido en Junio 18, 2023, de https://enix.io/en/blog/cache-image-docker-kubernetes/

[11] *Use image caches to accelerate the creation of pods - Container Service for Kubernetes*. (2022, Mayo 19). Alibaba Cloud. Accedido en Junio 23, 2023, desde https://www.alibabacloud.com/help/en/container-service-for-kubernetes/latest/use-the-image-cache-crd-to-accelerate-the-pod-creation-process

[12] Bekker, S. (2019, Noviembre 21). *New 'Project Teleport' Speeds Azure Container Creation*. Visual Studio Magazine. Accedido en Junio 23, 2023, desde https://visualstudiomagazine.com/articles/2019/11/21/project-teleport.aspx

[13] Lasker, S. (2019, Octubre 29). *Azure Container Registry Adds Teleportation – Steve Lasker*. Steve Lasker. Accedido en Junio 23, 2023, desde https://stevelasker.blog/2019/10/29/azure-container-registry-teleportation/

[14] A. Kumari, B. Sahoo and R. K. Behera, "Mitigating Cold-Start Delay using Warm-Start Containers in Serverless Platform," *2022 IEEE 19th India Council International Conference (INDICON)*, Kochi, India, 2022, pp. 1-6, doi: 10.1109/INDICON56171.2022.10040220.

[15] *Configuring provisioned concurrency - AWS Lambda*. (n.d.). AWS Documentation. Accedido en Junio 25, 2023, desde https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html

[16] *Kube-fledged*:在*Kubernetes* 中缓存容器镜像. (2021, October 9). 知乎专栏. Accedido en Junio 26, 2023, desde https://zhuanlan.zhihu.com/p/419779028

[17] *containerd/stargz-snapshotter: Fast container image distribution plugin with lazy pulling*. (n.d.). GitHub. Accedido en Junio 26, 2023, desde https://github.com/containerd/stargz-snapshotter

[18] *uber/kraken: P2P Docker registry capable of distributing TBs of data in seconds*. (n.d.). GitHub. Accedido en Junio 26, 2023, desde https://github.com/uber/kraken

[19] *ContainerSolutions/ImageWolf: Fast Distribution of Docker Images on Clusters*. (n.d.). GitHub. Accedido en Junio 26, 2023, desde https://github.com/ContainerSolutions/ImageWolf

[20] Rehemägi, T. (2021, August 12). *How to Solve the Problem of Cold Starts in 'Serverless' Systems*. HackerNoon. Accedido en Junio 26, 2023, desde https://hackernoon.com/how-to-solve-the-problem-of-cold-starts-in-serverless-systems/

[21] *How does language, memory and package size affect cold starts of AWS Lambda?* (n.d.). Pluralsight. Accedido en Junio 26, 2023, desde https://www.pluralsight.com/resources/blog/cloud/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda

[22] Daly, J. (n.d.). *jeremydaly/lambda-warmer: A module to optimize AWS Lambda function cold starts*. GitHub. Accedido en Junio 26, 2023, desde https://github.com/jeremydaly/lambda-warmer

[23] Amazon Web Services. (n.d.). *„ ”*. SlideShare. Accedido en Junio 27, 2023, desde https://www.slideshare.net/AmazonWebServices/become-a-Serverless-black-belt-optimizing-your-Serverless-applications-srv401-reinvent-2017

[24] Emad Heydari Beni, Eddy Truyen, Bert Lagaisse, Wouter Joosen, and Jordy Dieltjens. 2021. Reducing cold starts during elastic scaling of containers in kubernetes. In Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC '21). Association for Computing Machinery, New York, NY, USA, 60–68. https://doi.org/10.1145/3412841.3441887

[25] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Slacker: Fast distribution with lazy docker containers. In 14th {USENIX } Conferencia sobre File and Storage Technologies ({FAST } 16). 181–195.

[26] Lin, Ping-Min & Glikson, Alex. (2019). Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach.

[27] Baldini, I. *et al.* (2017). Serverless Computing: Current Trends and Open Problems. In: Chaudhary, S., Somani, G., Buyya, R. (eds) Research Advances in Cloud Computing. Springer, Singapore. https://doi.org/10.1007/978-981-10-5026-8_1

[28] *Fine-tuning the cold-start in OpenFaaS*. (2023, March 28). OpenFaaS. Accedido en Julio 10, 2023, desde https://www.openfaas.com/blog/fine-tuning-the-cold-start/

[29] GitHub Issue: Improve cold start times. https://github.com/apache/openwhisk-runtime-dotnet/issues/34

[30] Tokunaga, K. (2020, Abril 22). *Startup Containers in Lightning Speed with Lazy Image Distribution on Containerd*. Medium. Accedido en Agosto 4, 2023, desde https://medium.com/nttlabs/startup-containers-in-lightning-speed-with-lazy-image-distribution-on-containerd-243d94522361

[31] *uber/kraken: P2P Docker registry capable of distributing TBs of data in seconds*. (n.d.). GitHub. Retrieved June 26, 2023, from https://github.com/uber/kraken

[32] Evento. (2023). En Wikipedia. https://es.wikipedia.org/wiki/Evento

[33] Función (programación). (2023). En Wikipedia. https://es.wikipedia.org/wiki/Función_(programación)

[34] Event-driven architecture. (2023). En Wikipedia. https://en.wikipedia.org/wiki/Event-driven_architecture

[35] Cold Start. (2023). En Wikipedia. https://en.wikipedia.org/wiki/Cold_start

[36] AWS Lambda. (2023). Amazon Web Services. https://aws.amazon.com/es/lambda/

[37] Géhberger, Dániel & Kovács, Dávid. (2022). Cooling Down FaaS: Towards Getting Rid of Warm Starts. 10.48550/arXiv.2206.00599.

[38] *Kubernetes documentation: Concepts*. (n.d.). Kubernetes. Accedido en Setiembre 6 del 2023, en https://kubernetes.io/docs/concepts/

[39] *About storage drivers*. (n.d.). Docker Docs. Accedido Septiembre 7 de 2023, desde https://docs.docker.com/storage/storagedriver/#images-and-layers

[40] *Docker overview*. (n.d.). Docker Docs. Accedido en Septiembre 7 de 2023, desde https://docs.docker.com/get-started/overview/

[41] *Dynamic Admission Control*. (2023, August 18). Kubernetes. Accedido September 7, 2023, from https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/

[42] P. Vahidinia, B. Farahani and F. S. Aliee, "Cold Start in Serverless Computing: Current Trends and Mitigation Strategies," 2020 International Conference on Omni-layer Intelligent Systems (COINS), Barcelona, Spain, 2020, pp. 1-7, doi: 10.1109/COINS49042.2020.9191377.

[43] Home | OpenFaaS - Serverless Functions Made Simple. Accedido en Setiembre 10 del 2023, desde https://www.openfaas.com/

[44] Apache OpenWhisk is a serverless, open source cloud platform. Accedido en Setiembre 10, 2023, desde https://openwhisk.apache.org/

[45] *grycap/imagemagick - Docker Image*. (n.d.). Docker Hub. Accedido en Setiembre 10, 2023, desde https://hub.docker.com/r/grycap/imagemagick