Graduado en Ingeniería del Software

Funciones como servicio en el contexto del Internet de las Cosas
Functions as a service in the context of the Internet of Things

Realizado por
Altair Bueno Calvente

Tutorizado por
Bartolomé Rubio Muñoz
Cristian Martín Fernández

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, junio 2023

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

Funciones como servicio en el contexto del Internet de las Cosas

Functions as a service in the context of the Internet of Things

Realizado por
Altair Bueno Calvente

Tutorizado por
Bartolomé Rubio Muñoz
Cristian Martín Fernández

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2023

Fecha defensa: junio de 2023

Abstract

In recent years, serverless computing has become a popular alternative to traditional Cloud computing. We examine the usability and developer experience of three open-source serverless platforms in the context of the Internet of Things (IoT): OpenFaaS, Fission, and OpenWhisk. We also share our experience developing a low-latency Distributed Deep Neural Network (DDNN) application using serverless computing. Our research indicates that these platforms are user-friendly and accelerate application deployment, but demand more resources and are not suitable for constrained devices. Moreover, we achieved a 55 % improvement over Kafka-ML's performance under load, a framework that lacks dynamic scaling support, illustrating the potential of serverless computing for IoT applications. Finally, we extend Kafka-ML with Fission, unlocking serverless computing.

Keywords: Serverless, DDNN, Cloud-to-Things continuum

Resumen

En los últimos años, la computación *serverless* se ha convertido en una alternativa popular a la computación Cloud tradicional. Examinamos la usabilidad y la experiencia de desarrollo en tres plataformas *serverless* de código abierto en el contexto del Internet de las Cosas (IoT): OpenFaaS, Fission y OpenWhisk. También compartimos nuestra experiencia desarrollando una aplicación de Redes Neuronales Profundas Distribuidas (DDNN) de baja latencia utilizando dichas plataformas. Nuestra investigación indica que estas plataformas son fáciles de usar y aceleran el despliegue de aplicaciones, pero requieren de más recursos y no son adecuadas para dispositivos limitados. Además, logramos una mejora del 55 % sobre Kafka-ML bajo carga, un framework que carece de escalado dinámico, ilustrando el potencial de la computación *serverless* para aplicaciones IoT. Por último, extendemos el framework Kafka-ML con Fission, desbloqueando la computación *serverless* en la plataforma.

Palabras clave: Serverless, DDNN, Cloud-to-Things continuum

Índice

1. Introducción **9**

1.1. Motivación 9

1.2. Objetivos 9

1.3. Metodología 10

1.4. Estructura de la memoria 10

2. Tecnologías **13**

2.1. Docker 13

2.2. Kubernetes 13

2.2.1. K3S 13

2.2.2. Helm 14

2.3. OpenFaaS 14

2.4. Fission 14

2.5. OpenWhisk 14

2.6. Python 15

2.7. Tensorflow 15

2.8. Jupyter Notebook 15

2.9. TypeScript 15

2.10. Deno 15

2.11. Apache Kafka 16

2.12. Kafka-ML 16

2.13. Raspberry Pi 16

2.14. Visual Studio Code 16

2.15. Git 17

2.16. GitHub 17

3. Introducción a las Funciones como Servicio **19**

3.1. OpenFaaS 19

3.1.1. Instalación 20
3.1.2. Creando nuestra primera función 21
3.2. Fission 22
3.2.1. Instalación 23
3.2.2. Creando nuestra primera función 23
3.3. OpenWhisk 24
3.3.1. Instalación 24
3.3.2. Creando nuestra primera función 24

4. Inferencia utilizando Redes Neuronales Profundas Distribuidas 27
4.1. Diseño y arquitectura 27
4.2. Requisitos funcionales 29
4.3. Requisitos no funcionales 30
4.4. Desarrollo e implementación en OpenFaaS 30
4.5. Desarrollo e implementación en Fission 31
4.6. Desarrollo e implementación en OpenWhisk 32
4.7. Integración con Kafka-ML 32

5. Evaluación 35
5.1. Entorno de pruebas 35
5.2. Resultados obtenidos 37
5.3. Interpretación de los resultados 38

6. Evaluación sin Edge 41
6.1. Resultados obtenidos 41
6.2. Interpretación de los resultados 43

7. Conclusiones y Líneas Futuras 45
7.1. Conclusiones 45
7.2. Líneas Futuras 46

Índice de figuras

1. Raspberry Pi modelo 3B+ 17
2. Modelo de la arquitectura para OpenFaaS, Fission y OpenWhisk 28
3. Diagrama de secuencia de nuestro sistema 28
4. Diagrama de secuencia del ciclo de vida de la función 33
5. Definición de los modelos en Kafka-ML 36
6. Diagrama de caja de los tiempos de respuesta por cada plataforma y capa . . . 38
7. Resultados obtenidos con un solo cliente 42
8. Resultados obtenidos con tres clientes 42
9. Resultados obtenidos con cinco clientes 43

1 Introducción

1.1. Motivación

Al contrario que los sistemas de computación tradicionales, como los sistemas cloud, la computación sin servidor, también conocida como función como servicio, presenta un cambio de paradigma en el que las aplicaciones y sus funciones son instanciadas y ejecutadas únicamente cuando éstas son invocadas. Esto no sólo permite liberar los recursos de las infraestructuras cuando no se requiera un uso de las aplicaciones, sino también un auto-escalado de las aplicaciones en momentos de necesidad, lo que conlleva a un uso eficiente de recursos. En el contexto del Internet de las Cosas (IoT), donde paradigmas con recursos limitados como la computación en el borde (*edge computing*) juegan un papel muy importante para el desarrollo de aplicaciones con requisitos de baja latencia, la computación sin servidor se presenta como una tecnología prometedora.

1.2. Objetivos

El presente trabajo pretende estudiar la adopción y las limitaciones de la computación sin servidor en este contexto. En primer lugar, se desarrollará una aplicación IoT para validar la tecnología, y posteriormente, a partir de los conocimientos adquiridos, se adaptará el framework Kafka-ML[1] del grupo de investigación ERTIS (Embedded Real-Time Systems) de la UMA para el uso de computación sin servidor. Kafka-ML es un proyecto de código abierto para la gestión completa del ciclo de vida de aplicaciones de aprendizaje automático a través de flujos continuos de datos. La perfecta integración de las mencionadas tecnologías permitirá el desarrollo de aplicaciones IoT y aprendizaje automático adaptadas a las necesidades de cada momento.

Los objetivos de este trabajo consisten en adaptar el framework Kafka-ML para su ejecu-

ción en plataformas de funciones como servicio, así como el estudio y evaluación de dichas plataformas. En concreto, los objetivos del trabajo incluyen:

- **Estudio y evaluación de tecnologías de código abierto de computación sin servidor**: Se analizarán las fortalezas y debilidades de plataformas como OpenWhisk, OpenFaaS y Fission
- **Diseño y desarrollo de una aplicación IoT de monitorización**: Desarrollo del software necesario para probar las plataformas de computación sin servidor y la adaptación del framework Kafka-ML
- **Adaptación del framework Kafka-ML**: Se adaptará el framework Kafka-ML del grupo de investigación ERTIS para que pueda ser ejecutado bajo demanda en las plataformas de computación sin servidor anteriormente descritas
- **Desarrollo de una aplicación IoT a partir del framework Kafka-ML adatado**: Se validará la solución a través de una aplicación, a modo de prueba de concepto, sobre el funcionamiento de Kafka-ML en dichas plataformas de computación sin servidor

1.3. Metodología

La metodología utilizada en el trabajo es la metodología ágil, basada en el desarrollo iterativo e incremental, donde los requisitos y soluciones evolucionan con el tiempo según la necesidad del proyecto.

Cada iteración ha tenido una duración de entre 2 y 3 semanas. En cada una de estas iteraciones, se han mantenido reuniones con los tutores de este Trabajo de Fin de Grado y el responsable del grupo ERTIS con objeto de valorar el progreso y planificar los siguientes pasos.

El uso de esta metodología nos proporciona flexibilidad frente a los cambios, permitiendo una buena adaptación a las circunstancias que puedan ocurrir durante el transcurso del trabajo.

1.4. Estructura de la memoria

La memoria está divida en 7 secciones: en la Sección 1 introducimos la motivación, objetivos y metodología de este trabajo. Posteriormente, en la Sección 2, describimos todas las

tecnologías utilizadas. La Sección 3 introduce las plataformas *serverless* evaluadas, haciendo hincapié en su instalación y facilidad de uso. La Sección 4 introduce nuestro sistema de inferencia distribuido basado en plataformas *serverless*, así como nuestra experiencia de desarrollo para cada cada plataforma. En la Sección 5, evaluamos el rendimiento de la implementación a través del *Cloud-to-Things continuum* (Edge, Fog, Cloud). En la Sección 6 evaluamos de nuevo el sistema desarrollado sin la capa Edge y comparamos su rendimiento con Kafka-ML. Por último, en la Sección 7, exponemos las conclusiones de este trabajo y las posibles líneas futuras.

2 Tecnologías

2.1. Docker

Docker [2] es una plataforma de software que permite a los desarrolladores crear, distribuir y ejecutar aplicaciones en contenedores. Un contenedor es un paquete liviano que incluye todo lo necesario para que una aplicación se ejecute, como el código, las librerías y las dependencias. Los contenedores de Docker han resultado esenciales para la realización de este trabajo, dado que gracias a su portabilidad pueden ejecutarse en cualquier sistema operativo y en cualquier infraestructura. Esto simplifica el paso de un entorno de desarrollo a un entorno de producción sin preocupaciones por las diferencias de configuración. Esto es especialmente útil para aplicaciones que se ejecutan en múltiples arquitecturas hardware.

2.2. Kubernetes

Kubernetes [3] es un sistema *open-source* para la automatización del despliegue, escalado y gestión de aplicaciones en contenedores. Diseñado originalmente por Google para solventar los problemas de escalabilidad de sus servicios, Kubernetes permite ejecutar miles de aplicaciones que escalan automáticamente sin la supervisión continua de un equipo de operaciones.

Actualmente, Kubernetes es un proyecto perteneciente a la fundación Linux, concretamente a la Cloud Native Computing Foundation (CNCF) [4, 5]. Además del proyecto original, han aparecido otros orquestadores compatibles con Kubernetes que presentan distintas capacidades y objetivos [6].

2.2.1. K3S

K3s [7] es una distribución de Kubernetes *open-source* diseñada para dispositivos IoT y Edge. Para lograr la máxima compatibilidad con estos dispositivos, K3s ha sido empaquetado en

un único ejecutable sin dependencias externas. Dicho ejecutable está disponible para múltiples arquitecturas.

A diferencia de otras distribuciones de Kubernetes, K3s no es un *fork*[1] de Kubernetes, sino un proyecto independiente. K3s ha sido certificado por CNCF como orquestador compatible con Kubernetes [6].

2.2.2. Helm

Helm [9] es una herramienta de gestión de paquetes para Kubernetes. Entre otras cosas, Helm simplifica la instalación y el mantenimiento de aplicaciones en Kubernetes, permitiendo desplegar aplicaciones complejas compuestas múltiples recursos de forma rápida, sencilla y reproducible.

2.3. OpenFaaS

OpenFaaS [10] es una plataforma de funciones como servicio (FaaS) de código libre basada en contenedores. Aparece destacada en el Cloud Native Landscape [11], siendo esta la plataforma de FaaS de código libre más famosa actualmente. Además de la versión *open-source*, OpenFaaS ofrece una versión de pago llamada OpenFaaS Pro que ofrece funciones adicionales y soporte comercial.

2.4. Fission

Fission [12] es una plataforma de FaaS de código abierto basado en Kubernetes. Actualmente, aparece destacada en el Cloud Native Landscape como la segunda plataforma *open-source* más famosa [11].

2.5. OpenWhisk

OpenWhisk [13] es una plataforma de FaaS de código abierto de la fundación Apache. La plataforma fue desarrollada originalmente por IBM y alimenta su plataforma *"IBM Cloud Functions"* [14].

[1] Un fork es un proyecto independiente que surge a partir de otro proyecto [8]

2.6. Python

Es un lenguaje de programación de alto nivel enfocado en la legibilidad y expresividad. El lenguaje es interpretado, con soporte a programación orientada a objetos y funcional, utiliza un recolector de basura para gestionar la memoria y tipado dinámico [15].

2.7. Tensorflow

Plataforma para el desarrollo de aplicaciones de aprendizaje automático. Desarrollado originalmente por Google, Tensorflow se ha posicionado como el framework de aprendizaje automático por excelencia. Sus librerías son accesibles a través de lenguajes como JavaScript, Python y C++, con soporte a multitud de arquitecturas, entre ellas amd64, aarch64 y CUDA [16].

2.8. Jupyter Notebook

Jupyter Notebook [17] es una aplicación web de código abierto que permite a los usuarios crear y compartir documentos que contienen código, ecuaciones, visualizaciones y texto. Los documentos de Jupyter Notebook se organizan en cuadernos (llamados *notebooks*) que contienen una combinación de celdas de código, celdas de texto y celdas de visualización.

2.9. TypeScript

TypeScript [18] es un lenguaje de programación de código abierto y tipado estáticamente desarrollado por Microsoft que se basa en JavaScript. TypeScript se basa en la sintaxis de JavaScript, pero agrega tipos de datos opcionales y otros elementos de programación orientada a objetos como enumeraciones, interfaces y módulos.

2.10. Deno

Deno [19] es un entorno de ejecución de JavaScript y TypeScript para ejecutar en el servidor, similar a NodeJS [20]. A diferencia de NodeJS, Deno ofrece herramientas de desarrollo integradas, permisos, soporte a paquetes de NPM y ejecución de TypeScript sin transpilado previo.

2.11. Apache Kafka

Apache Kafka [21] es una plataforma de streaming de datos de código abierto, desarrollada por la Apache Software Foundation. Kafka se utiliza para transmitir datos en tiempo real y procesar grandes cantidades de datos, donde los consumidores se pueden suscribir a temas donde los productores publican datos.

2.12. Kafka-ML

Kafka-ML [1] es un framework para la gestión de modelos de *Machine Learning* (ML) de Tensorflow y PyTorch en Kubernetes. La plataforma permite diseñar, entrenar y desplegar modelos de ML. Los datos de entrenamiento e inferencia se proporcionan a través de Apache Kafka, conectando directamente la plataforma con los flujos de datos generados por los dispositivos IoT.

2.13. Raspberry Pi

Raspberry Pi [22] es una serie de computadoras de placa única de bajo coste y tamaño reducido, desarrollada por la Raspberry Pi Foundation. Aunque originalmente se diseñó como una plataforma de enseñanza de informática para estudiantes, ha evolucionado hasta convertirse en un dispositivo popular para proyectos de automatización del hogar, robótica, servidores web, dispositivos IoT y muchas otras aplicaciones. Para este trabajo, el modelo de Raspberry que hemos utilizado es la Raspberry Pi 3B+ (Figura 1).

2.14. Visual Studio Code

Visual Studio Code [23] es un editor de código de código libre políglota desarrollado por Microsoft. El editor ofrece soporte para lenguajes como JavaScript, TypeScript, JSON y Markdown. A través de extensiones, el editor se puede extender para soportar otros lenguajes, funciones o integraciones con otras herramientas de desarrollo. Para el desarrollo de este trabajo, hemos instalado las extensiones *Deno*, *python* y *pylance*.

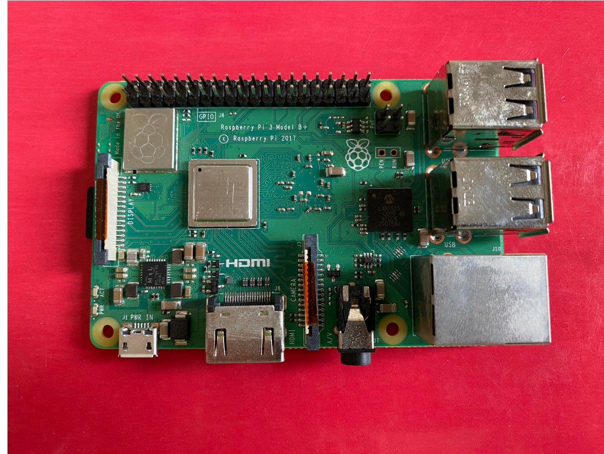


Figura 1: Raspberry Pi modelo 3B+

2.15. Git

Diseñado originalmente por Linus Torvalds para gestionar el código fuente de Linux, Git [24] es un sistema de control de versiones gratuito de código libre diseñado para gestionar proyectos de cualquier tamaño de forma eficiente y rápida. Actualmente, es el sistema de control de versiones más utilizado y cuenta con el apoyo de grandes empresas como Microsoft, Google y Twitter.

2.16. GitHub

GitHub [25] es una compañía que ofrece servicios para el desarrollo de software y control de versiones a través de Git. Adquirida en 2018 por Microsoft [26], GitHub es actualmente la plataforma de *hosting* con mayor número de proyectos de código libre.

Además de sus servicios de control de versiones, hemos utilizado las siguientes herramientas de la plataforma: Control de versiones mediante Git, GitHub Actions, GitHub Packages y GitHub Projects.

3 Introducción a las Funciones como Servicio

En los últimos años, las Funciones como Servicio (FaaS) han demostrado ser una alternativa válida a la computación en la nube tradicional. Su naturaleza sin estado les permite escalar horizontalmente en base a eventos, permitiendo un uso más flexible de los recursos [27]. Esto facilita el trabajo de los desarrolladores, quienes se pueden despreocupar de la infraestructura y centrarse en resolver problemas.

Muchas aplicaciones del Internet de las Cosas (IoT) generan cantidades masivas de datos que necesitan ser procesadas de forma eficiente y flexible. Es esencial disponer de soluciones Cloud y Fog/Edge que puedan adaptarse y reaccionar a flujos inesperados de datos. Consideramos que las funciones *serverless* son una solución prometedora para simplificar el desarrollo, despliegue y mantenimiento de aplicaciones IoT a través del *Cloud-to-Things continuum* [28].

Aunque la mayoría de proveedores Cloud ofrecen plataformas *serverless*, estas suelen ser de código privado y pueden dar lugar a dependencias con el proveedor [29, 30]. En esta sección, haremos una introducción a las tres plataformas FaaS *open-source* analizadas, incluyendo nuestra experiencia en su instalación y su facilidad a la hora de crear funciones básicas.

3.1. OpenFaaS

OpenFaaS es una plataforma de computación sin servidor basada en contenedores que aparece listada en la *Cloud Native Landscape*. Su código se encuentra licenciado bajo la permisiva licencia MIT y actualmente es la plataforma *serverless* con más estrellas en GitHub, con 22k estrellas [10].

OpenFaaS ofrece dos versiones: OpenFaaS Community Edition (CE) y OpenFaaS Pro. OpenFaaS Pro es una versión de pago que extiende las funciones de la versión CE para ofrecer algunas características extra, así como soporte comercial. Una de las diferencias más importantes entre ambas versiones es el auto escalado de las funciones. La versión CE utiliza *"Legacy scaling"*, el cual no permite reducir el número de instancias a cero . Además, OpenFaaS indica que el auto escalado de la versión CE esta pensado solo para desarrollo o para uso interno, no para aplicaciones comerciales [31].

Una de las principales características de OpenFaaS son las *plantillas (templates)*. Estas permiten a los desarrolladores crear proyectos de OpenFaaS en cualquier lenguaje de programación soportado de forma rápida y sencilla. OpenFaaS ofrece una gran variedad de plantillas de distintos lenguajes, incluyendo algunos populares como Python, JavaScript, Ruby y Java. Además de las plantillas oficiales, existe un gran número de plantillas mantenidas por la comunidad que permiten a los desarrolladores crear funciones en otros lenguajes no soportados oficialmente [32]. Por si fuera poco estas plantillas suelen tener variantes que o bien incluyen librerías extra o bien sustituyen la imagen base de Docker, ofreciendo aún más posibilidades.

Las funciones pueden ser invocadas de múltiples formas, como peticiones HTTP y eventos de MQTT y S3. Además, la plataforma también soporta peticiones asíncronas, en las que los eventos son almacenados en una cola NATS de tipo FIFO para su posterior ejecución. Un detalle a tener en cuenta sobre las peticiones asíncronas es que estas se ejecutan secuencialmente, incluso si estas son destinadas a funciones distintas.

3.1.1. Instalación

OpenFaaS ofrece varias opciones para su despliegue. La plataforma se puede desplegar en orquestadores de contenedores como Kubernetes, K3s y OpenShift, o incluso una versión reducida en máquinas con menos recursos a través de `faasd` [33]. Para nuestro uso, hemos optado por instalar la plataforma completa en Kubernetes y K3s utilizando Helm [9]. La instalación fue sencilla y no encontramos ningún problema, incluso para nuestro cluster de K3s sobre ARM. La instalación incluye componentes como Prometheus y NATS, dando lugar a una instalación completa. En cuestión de minutos, conseguimos desplegar nuestra primera función básica en un cluster completamente operativo.

Además del panel del control, la guía de instalación [34] incluye instrucciones para instalar

la herramienta de línea de comandos (CLI) `faas-cli`. Esta utilidad está disponible para su descarga en las plataformas Linux, Windows y macOS. Aunque este CLI no es esencial, se recomienda su instalación ya que simplifica el desarrollo de las funciones.

3.1.2. Creando nuestra primera función

La documentación de OpenFaaS [35] incluye múltiples tutoriales y recursos para facilitar los primeros pasos en la plataforma. Por ejemplo, el tutorial *"Your first OpenFaaS Function with Python"* [36] es sencillo y explica todo lo necesario para desplegar una función en Python con dependencias. El proceso de desarrollo de las funciones normalmente involucra los siguientes pasos:

1. **Crear el proyecto**: Utilizando las plantillas, el desarrollador puede crear una nueva función en cualquier lenguaje de los soportados oficialmente, o bien alguno de los ofrecidos por la comunidad.

2. **Modificar el código fuente**.

3. **Actualizar el archivo de despliegue**: Cada función incluye un fichero YAML que describe como `faas-cli` desplegará la función. El desarrollador puede activar ciertas opciones al modificar el fichero, como las variables de entorno, secretos y argumentos de Docker.

4. **Desplegar la función**: Utilizando el fichero de despliegue, `faas-cli` se encargará de crear la imagen de Docker de nuestra función, que será almacenada en un registro de Docker para ser posteriormente instanciada.

OpenFaaS ofrece una experiencia de desarrollo muy sencilla, ya que la plataforma abstrae las tecnologías con las que trabaja. El desarrollador no necesita tener conocimientos sobre Docker o Kubernetes, ni siquiera sobre la plataforma en si mismo. Un desarrollador puede crear un proyecto y desplegar una función con solo ejecutar un par de comandos, reduciendo la fricción entre el desarrollador y la plataforma.

3.2. Fission

Otra plataforma que aparece en el *Cloud Native Landscape* es Fission. Esta plataforma, basada en en Kubernetes, es la segunda con más estrellas de GitHub, con 7.4K estrellas [12]. El código se encuentra licenciado bajo la licencia Apache 2.0. Esta plataforma destaca por ofrecer los llamados *entornos (environments).*

Los entornos son las partes de Fission que permiten la creación de funciones en otros lenguajes de programación. Dichos entornos están compuestos de un contenedor con un servidor HTTP, y un cargador dinámico (llamado *fetcher*) que inyecta la función para su ejecución. Algunos entornos también incluyen contenedores llamados *constructores (builders)* encargados de compilar e instalar las dependencias de nuestra función [37]. Los entornos están disponibles con las siguientes estrategias:

- **PoolManager**: Fission mantiene un conjunto de pods "calientes" del entorno seleccionado. Cuando una petición llega, el contenedor *fetcher* descarga la función y lo inyecta en el entorno. Este pod será utilizado para las siguientes peticiones, y será eliminado después de un tiempo sin actividad. Esta estrategia es ideal para funciones que tienen poca duración y son sensibles a latencias altas.

- **NewDeploy**: Crea un despliegue de Kubernetes con un servicio y un *HorizontalPodAutoscaler* (HPA) [38]. Esto permite el escalado automático de la función y balanceo de carga de las peticiones entre los pods. Cuando la función experimenta una subida en las peticiones, el servicio ayuda a distribuirlas entre todos los pods de la función. El HPA escala las replicas del despliegue basándose en las condiciones impuestas por el usuario. Si no hay peticiones por un tiempo determinado, los pods sin actividad serán eliminados. Esta estrategia aumenta las latencias debidas a los *cold starts*, pero permite a la función gestionar un trafico masivo [39].

Fission tiene una característica que lo diferencia de otras plataformas serverless: Sus entornos son pods de Kubernetes completos. Esto le permite utilizar todos los recursos disponibles en el orquestador como los secretos, volúmenes y GPUs. Esto proporciona a los desarrolladores mucha flexibilidad y les permite crear funciones complejas que se aprovechan del extenso ecosistema de Kubernetes.

3.2.1. Instalación

Fission puede ser instalado utilizando Helm u objetos de Kubernetes [40], aunque la instalación mediante Helm ofrece más opciones. Esto es importante, ya que la instalación base de Fission solo incluye los componentes esenciales para desarrollar y probar funciones. Esta instalación incluye un panel de control mucho más pequeño, pero requiere que el operador active cada característica manualmente.

La guía de instalación [41] también incluye instrucciones sobre como instalar `fission`, una herramienta de CLI disponible en Linux, Windows y macOS. El CLI es esencial, ya que es la única forma de operar Fission.

3.2.2. Creando nuestra primera función

La documentación de Fission [42] es completa y describe todas las funciones principales de la plataforma. El equipo de desarrollo ha creado una colección de ejemplos que incluye todos los lenguajes que soportan oficialmente, lo que simplifica enormemente el desarrollo de las funciones. El flujo de trabajo normalmente involucra las siguientes fases:

1. **Elegir el entorno adecuado**: Cada función requiere un entorno en el que será ejecutado. Se pueden crear tantos entornos como sean necesarios, pero cada función solo podrá ejecutarse en un entorno a la vez.

2. **Crear el proyecto**: Cada entorno incluye su propio constructor, que se encarga de compilar e instalar las dependencias. El desarrollador es responsable de leer la documentación y crear el proyecto de acuerdo con los requisitos del constructor.

3. **Modificar el código fuente**.

4. **Desplegar el entorno y la función**: El CLI `fission` es utilizado para empaquetar el código fuente y enviarlo al constructor. Este se encargará de crear el archivo final que será cargado por el *fetcher*. Por defecto Fission no crea ningún *trigger* de las funciones, por lo que la función solo estará disponible para ejecución a través del CLI.

Para facilitar la tarea de despliegue, Fission ofrece *Kubernetes Custom Resources Definition* (CRD) [43] que son aplicados por el CLI al ejecutar `fission spec apply`. Una ventaja de

usar las especificaciones es que los entornos pueden ser extendidos utilizando *PodSpecs* de Kubernetes, permitiendo el acceso a volúmenes, variables de entorno y contenedores sidecar.

El enfoque único de Fission para las funciones como servicio tiene una curva de aprendizaje más empinada. Los desarrolladores deben comprender las bases de los entornos antes de crear las funciones, así como leer las instrucciones sobre el mismo para saber como estructurar el proyecto. No obstante, Fission ofrece, a través de las especificaciones, una interfaz familiar a aquellos desarrolladores que estén ya familiarizados con Kubernetes.

3.3. OpenWhisk

OpenWhisk es una plataforma de FaaS de la fundación Apache, licenciada bajo la licencia Apache 2.0. Tiene el menor número de estrellas en GitHub de todas las plataformas analizadas, con 5.9k [13]. OpenWhisk es la única plataforma desplegada por un proveedor cloud, ya que es la tecnología detrás de IBM Cloud Functions [14].

3.3.1. Instalación

La plataforma puede ser desplegada de muchas formas: Kubernetes, Docker (a través de Docker Compose), Ansible y Vagrant. Esto otorga a OpenWhisk una gran versatilidad, ya que la plataforma mantiene las misma funcionalidad independientemente del método de instalación. Para nuestro caso, elegimos instalar OpenWhisk en Kubernetes utilizando Helm.

La instalación fue complicada. La guía de instalación [44] es confusa, incompleta y desactualizada. No obstante, tras varios intentos, conseguimos su instalación con la ayuda de un documento que contiene algunos errores comunes, disponible en su repositorio [45].

La guía de instalación también incluye instrucciones sobre como instalar `wsk`, un CLI disponible en Linux, Windows y macOS utilizado para gestionar las funciones. Este CLI no es obligatorio, ya que la plataforma puede ser gestionada a través de peticiones a su API REST.

3.3.2. Creando nuestra primera función

La documentación de OpenWhisk incluye ejemplos mínimos de como desplegar funciones básicas en todos los lenguajes que soportan oficialmente, como JavaScript, Go, Python y Java. Estas funciones suelen estar compuestas por un único fichero fuente que se envía al panel

de control. Funciones más avanzadas pueden ser creadas, pero están sujetas a instrucciones especificas para cada lenguaje. El proceso de desarrollo de una función sigue los siguientes pasos:

- **Crear el proyecto**: El desarrollador es el encargado de crear la estructura del proyecto, ya que el CLI `wsk` carece de esta funcionalidad.
- **Modificar el código fuente**
- **Desplegar la función**: Dado que OpenWhisk no ofrece utilidades para compilar la función, el desarrollador es el encargado de compilar e instalar cualquier dependencia que la función necesite.

4 Inferencia utilizando Redes Neuronales Profundas Distribuidas

Una de las aplicaciones IoT que se podrían beneficiar de la flexibilidad ofrecida por las FaaS es la inferencia de modelos de Redes Neuronales Profundas (DNN). Las DNN son computacionalmente costosas y, a menudo, requieren una gran cantidad de recursos o dispositivos hardware especializados (p.ej. GPUs) que no están disponibles en aquellos dispositivos con menos recursos. Si distribuimos la DNN (DDNN) sobre el *Cloud-to-Things continuum*, logramos generar respuestas lo más cerca posible de donde se produce la información. Esto nos permite reducir los tiempos de respuesta [46] y cumplir con los requisitos de estos dispositivos menos potentes. Mediante el uso de FaaS, esperamos alcanzar niveles similares de latencia en DDNN a frameworks como Kafka-ML [47], pero con escalado horizontal automático y una experiencia de desarrollo mejorada. En esta sección introduciremos la arquitectura de nuestro sistema de DDNN sobre FaaS, así como nuestra experiencia implementando dicho sistema en cada una de las plataformas previamente analizadas.

4.1. Diseño y arquitectura

Nuestro sistema esta compuesto por tres capas distribuidas a lo largo del *Cloud-to-Things continuum* (Edge, Fog y Cloud). El sistema se basa en el anterior trabajo de Torres et al. [47], donde los autores utilizan Apache Kafka para las comunicaciones. Las capas Edge y Fog pre-

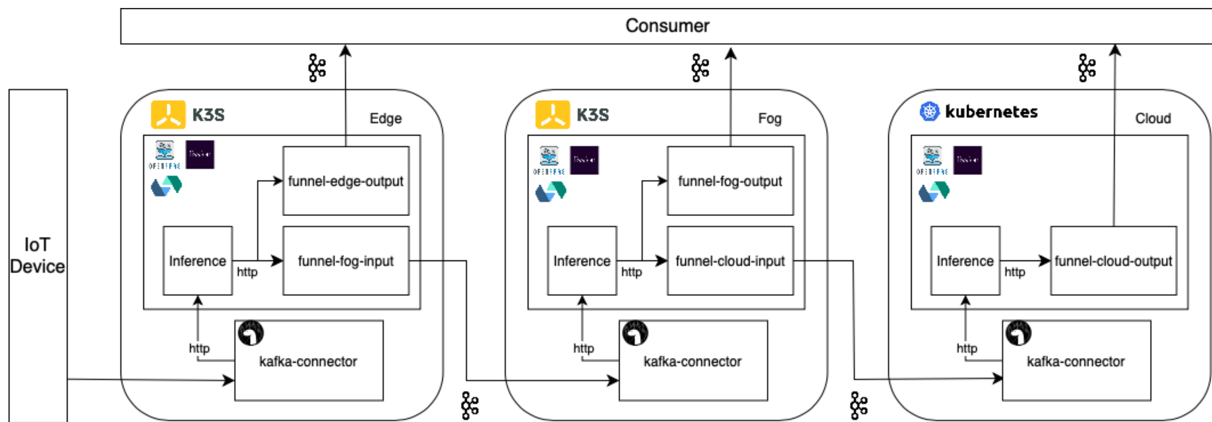


Figura 2: Modelo de la arquitectura para OpenFaaS, Fission y OpenWhisk

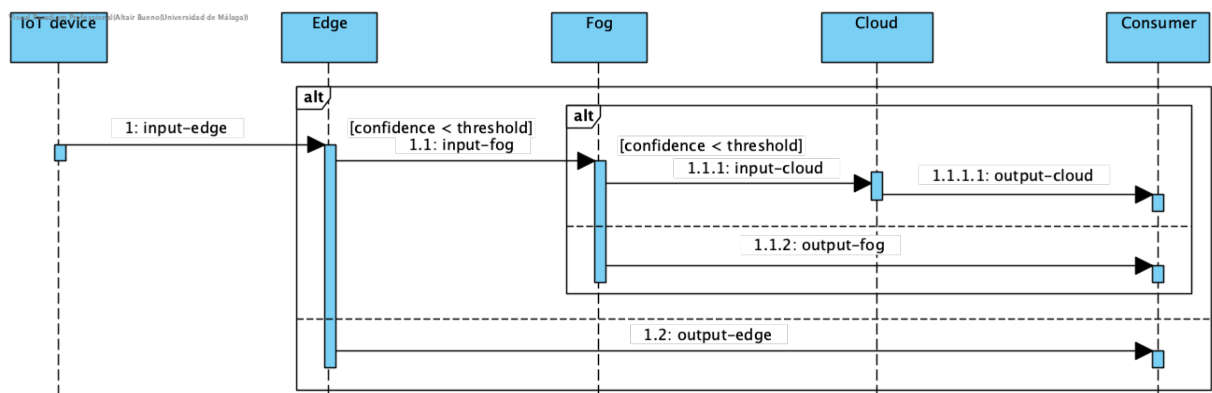


Figura 3: Diagrama de secuencia de nuestro sistema

sentan dos salidas: la primera, conocida como *early exit*, se utiliza para generar las predicciones intermedias de cada capa con el objetivo de reducir la latencia, mientras que la segunda se utiliza para elevar la inferencia a la capa superior. Cuando la predicción generada por una capa supera un umbral establecido esta se emite a través de la salida *early exit*, de lo contrario, esta se transmite a la capa superior. En la Figura 2 podemos ver el diseño de nuestro sistema para OpenFaaS, Fission y OpenWhisk, mientras que la Figura 3 ilustra la comunicación entre estas capas mediante un diagrama de secuencia.

Sin embargo, las FaaS de estas plataformas no están diseñadas para producir múltiples salidas. Para solucionarlo, existen varias alternativas: la primera opción sería conectar la función encargada de la inferencia directamente con Kafka. De esta forma, al producirse el resultado, la función lo escribiría en el tópico correspondiente. No obstante, este diseño presenta ciertas limitaciones en escalabilidad, ya que la conexión y desconexión con Kafka al escalar la función podría sobrecargar brokers de Kafka. Por esa razón, optamos por separar el trabajo en dos

funciones distintas. La función de inferencia (*inference*) se limita solo a producir el resultado, y delega la tarea de escritura en otra función (*funnel*). De esta forma, conseguimos que ambas funciones escalen de forma independiente.

Para la conexión entre Kafka y nuestro sistema, primero experimentamos con los conectores que cada una de las plataformas ofrecen. En el contexto de las FaaS, un conector es una aplicación encargada de extraer los valores de una fuente de datos y remitirlos por HTTP a la correspondiente función. En el caso de OpenFaaS, descubrimos que el conector oficial está limitado a los clientes de la versión OpenFaaS Pro, por lo que no pudimos utilizarlo. Por otro lado, Fission ofrece un conector *open-source* pero este presenta un comportamiento inestable. Observamos que al re-desplegar las funciones (tras modificar el código fuente o el modelo) dicho conector desaparecía de Kubernetes. Para hacer que el conector volviese a funcionar, era necesario reinstalar la plataforma.

Decidimos crear nuestro propio conector *kafka-to-http* y utilizarlo con todas las plataformas. Este conector está desarrollado en TypeScript y se ejecuta sobre Deno. Ya que Deno está diseñado con operaciones de E/S en mente, consideramos que es lo suficientemente rápido para la tarea.

Finalmente, sobre el formato utilizado para la transmisión, decidimos utilizar JSON. A pesar de que existen otros formatos más rápidos o más compactos, JSON ofrece una flexibilidad que otros formatos no ofrecen. Es un formato de serialización legible lo cual facilita la tarea de búsqueda y solución de fallos que podamos encontrar en el sistema.

4.2. Requisitos funcionales

Los requisitos funcionales son declaraciones claras y detalladas de las funcionalidades y capacidades que un sistema debe tener para satisfacer las necesidades y objetivos del usuario.

RF.1. El sistema realizará inferencia sobre los datos de entrada.

RF.2. El sistema podrá cargar modelos distribuidos (modelos con capas de salida intermedios) de TensorFlow.

RF.3. El sistema leerá datos de entrada de un tópico de Kafka.

RF.4. El usuario podrá especificar los brokers para el tópico de entrada de Kafka.

RF.5. El usuario podrá especificar el tópico de entrada de Kafka.

RF.6. El sistema escribirá el resultado en un tópico de Kafka si supera cierto nivel de certeza o no es un modelo distribuido.

RF.7. El usuario podrá especificar los brokers para la salida y salida parcial de Kafka.

RF.8. El usuario podrá especificar el tópico de salida y salida parcial de Kafka.

RF.9. El sistema escribirá el resultado parcial en un tópico de Kafka si y solo si se trata de un modelo distribuido que no supera cierto nivel de certeza.

RF.10. El usuario podrá especificar el nivel de certeza en modelos distribuidos.

RF.11. El usuario podrá especificar el número de capas en los que se distribuirá el sistema.

4.3. Requisitos no funcionales

Los requisitos no funcionales son criterios y restricciones que se aplican al sistema, pero que no están relacionados directamente con las funcionalidades específicas que debe cumplir.

RnF.1. El sistema se basará en FaaS para su funcionamiento.

RnF.2. El sistema será capaz de operar con normalidad bajo picos de carga.

RnF.3. El sistema debe ser ejecutable en arquitecturas `amd64` y `arm64`.

RnF.4. El sistema debe ser accesible a desarrolladores que no disponen de conocimientos sobre Kubernetes.

RnF.5. El sistema podrá cargar modelos de TensorFlow.

4.4. Desarrollo e implementación en OpenFaaS

Para implementar nuestra función de inferencia buscamos una plantilla de TensorFlow o PyTorch para OpenFaaS. Descubrimos que no existe ninguna plantilla oficial o de la comunidad para dichos frameworks, por lo que decidimos crear la nuestra basada en TensorFlow para las arquitecturas `amd64` y `arm64`. Estas están basadas en las plantillas oficiales `python3-http` [48].

No obstante, no pudimos crear una variante de TensorFlow GPU, ya que OpenFaaS todavía no soporta dispositivos de Kubernetes [49].

El código de nuestra función de inferencia se compone de un único fichero Python y el modelo de TensorFlow ya entrenado. Aunque podríamos haber creado un sistema de almacenamiento externo, como un almacenamiento de objetos S3, decidimos desechar la idea para simplificar nuestro sistema y reducir los *cold starts*. Por otra parte, la función *funnel* también está compuesta de un único fichero Python y esta se ejecuta sobre la plantilla oficial `python3-http` de OpenFaaS.

El despliegue de las funciones fue sencillo en nuestros clusters basados en `amd64`. Sin embargo, en nuestro cluster de Raspberry Pi (arquitectura `arm64`), encontramos problemas a la hora de crear las imágenes de Docker. Para el desarrollo de este trabajo, utilizamos un repositorio de imágenes local con un certificado TLS firmado por nosotros mismos. Dicho certificado no era reconocido por Buildx, el sistema utilizado por `faas-cli` para compilar las imágenes de Docker para otras arquitecturas. Aunque Buildx presenta opciones para añadir certificados, `faas-cli` no ofrece esa opción. Tras intentar múltiples alternativas y consultar a los creadores de OpenFaaS [50], decidimos utilizar una Raspberry Pi para compilar las imágenes directamente. De esta forma, evitábamos el uso de Buildx y logramos el despliegue en el cluster de Raspberry Pi.

4.5. Desarrollo e implementación en Fission

Al igual que OpenFaaS, Fission no ofrece entornos de TensorFlow o PyTorch. Creamos nuestros propios entornos y constructores de TensorFlow para `amd64` y `arm64`, ambos basados en los entornos oficiales `python-3.10` de Fission [51]. Dicho entorno utiliza solo la CPU, aunque podríamos crear uno que utilizase la GPU para la inferencia. Esto se debe a que Fission permite el acceso a dispositivos de Kubernetes.

Tras crear el entorno, pudimos recrear la función de inferencia en Fission, basándonos en la versión creada previamente para OpenFaaS. Por otra parte, la función *funnel* se ejecuta sobre el entorno `python-3.10` de Fission. Ambas funciones se comportan igual que sus versiones de OpenFaaS.

El despliegue de ambas funciones fue extremadamente sencillo. La plataforma se encarga de automatizar todo el ciclo de vida de las funciones, desde el código fuente hasta Pod. Las

dependencias de nuestro código fueron instaladas automáticamente por los constructores y almacenadas en el propio gestor de paquetes de Fission.

4.6. Desarrollo e implementación en OpenWhisk

Para OpenWhisk, primero probamos a crear una función básica en Python con dependencias, en concreto la famosa librería *requests* para hacer peticiones HTTP. Siguiendo el tutorial ofrecido por OpenWhisk "Python Packages in OpenWhisk"[52], encontramos múltiples dificultades por el camino. A diferencia de las demás plataformas, OpenWhisk requiere que el desarrollador sea el encargado de gestionar la instalación de dependencias y compilar su código fuente. En el caso de nuestra función en Python, teníamos que utilizar el CLI de Docker para generar un entorno virtual de Python con las dependencias, y posteriormente, enviar dicho entorno con nuestro código a OpenWhisk. Tras lograrlo, nos encontramos un mensaje de error críptico que no pudimos solucionar. Repetimos el procedimiento en múltiples ocasiones sin éxito, por lo que decidimos darnos por vencido y no implementar nuestro sistema en OpenWhisk.

4.7. Integración con Kafka-ML

En nuestras pruebas iniciales pudimos comprobar que Fission presenta el comportamiento más estable a la hora de escalar funciones, consideramos que es la mejor plataforma para ser integrada en Kafka-ML, concretamente con su backend.

No obstante, debido a que nuestro sistema hace uso de varias funciones y configuraciones, su despliegue total es complicado. Esto contrasta con la simplicidad esperada por las FaaS. Por ello, limitamos la integración a inferencia de DNN como primera toma de contacto.

Decidimos crear un entorno de Fission distinto para esta tarea. Este se basa en el entorno de TensorFlow que creamos anteriormente, pero con algunos cambios. En primer lugar, hemos eliminado el constructor, ya que las funciones descargan el modelo ya entrenado de Kafka-ML. Por otra parte, hemos eliminado todas las librerías que no eran necesarias para la inferencia, reduciendo así el tamaño de la imagen.

Este entorno que hemos desarrollado contiene todo lo necesario para descargar y ejecutar los modelos de TensorFlow de Kafka-ML, y solo requiere de configuración para su funciona-

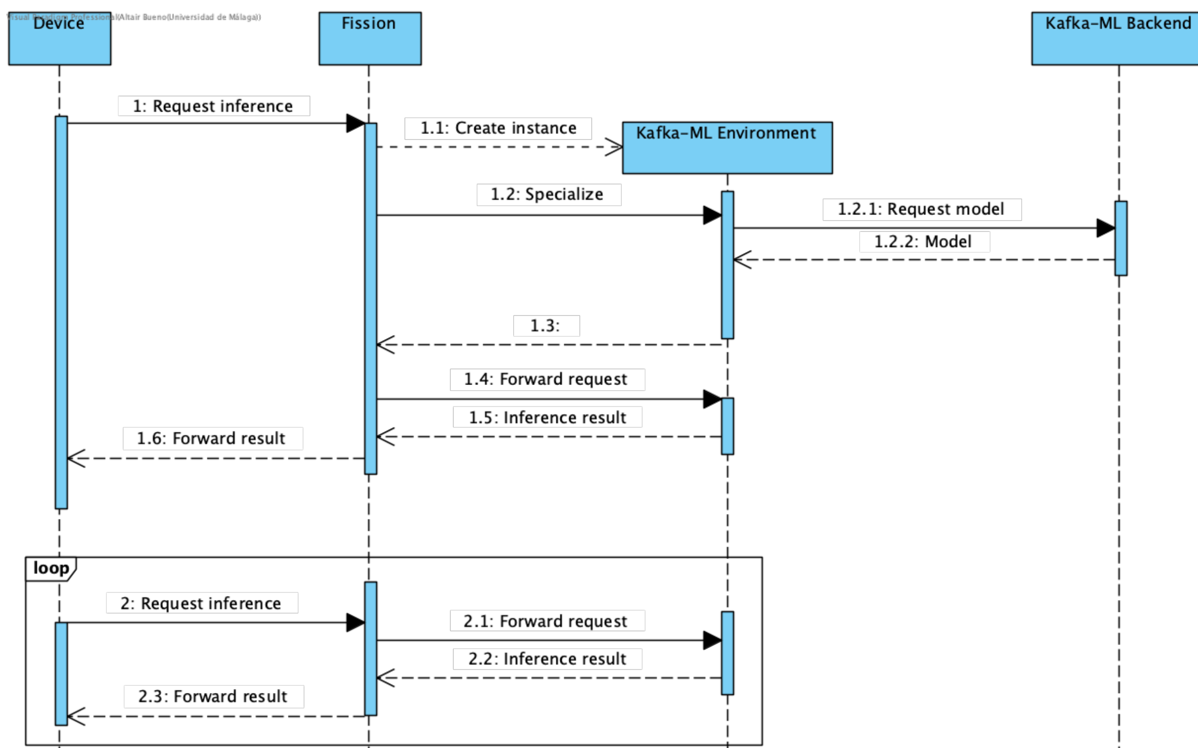


Figura 4: Diagrama de secuencia del ciclo de vida de la función

miento. En concreto, el entorno necesita la URL del backend de Kafka-ML y el identificador del modelo a utilizar. Estos valores serán proporcionados como código fuente de la función (en formato JSON) cuando Fission la instancie. En el Listing 1 podemos ver un ejemplo de este fichero de configuración, donde la propiedad "backend" es la URL del backend de Kafka-ML y la propiedad "model" es el identificador del modelo a utilizar.

```
{
  "backend": "http://backend.kafkaml:8000",
  "model": "1"
}
```

Listing 1: Ejemplo de configuración de un modelo

En la Figura 4 podemos observar el ciclo de vida de la función. Cuando llega una petición, Fission o bien instanciará un nuevo entorno (NewDeploy) o elegirá un entorno de los disponibles (PoolManager). Posteriormente, la plataforma especializará dicho entorno enviando el código fuente, que en nuestro caso es solo el fichero de configuración. El entorno se comunicará con Kafka-ML para obtener el modelo y lo cargará en TensorFlow. Una vez haya acabado

la fase de especialización, la función estará lista para procesar peticiones. Fission trasladará la petición original a la función y devolverá el resultado al remitente. Las siguientes peticiones que esta instancia procese serán más rápidas, ya que la función mantendrá el modelo cargado y no será necesario comunicarse con Kafka-ML.

Para conservar toda la funcionalidad ofrecida por Fission con versiones futuras, hemos optado por no integrar el despliegue en el frontend de Kafka-ML. La razón de esta decisión es que Fission no garantiza una API estable para el despliegue, por lo que nuestra integración podría dejar de funcionar en versiones futuras. Es por ello que recomendamos realizar el despliegue a través de su CLI. En el Listing 2 podemos ver un ejemplo de despliegue de una función que toma el fichero "config.json"de la carpeta actual como código fuente.

```
fission env create --image kafka-ml-env --name kafka-ml-env \
    --poolsize=0 --version=3
fission fn create --name mnist --env kafka-ml-env \
    --deployarchive config.json --entrypoint config.json \
    --executortype newdeploy
```

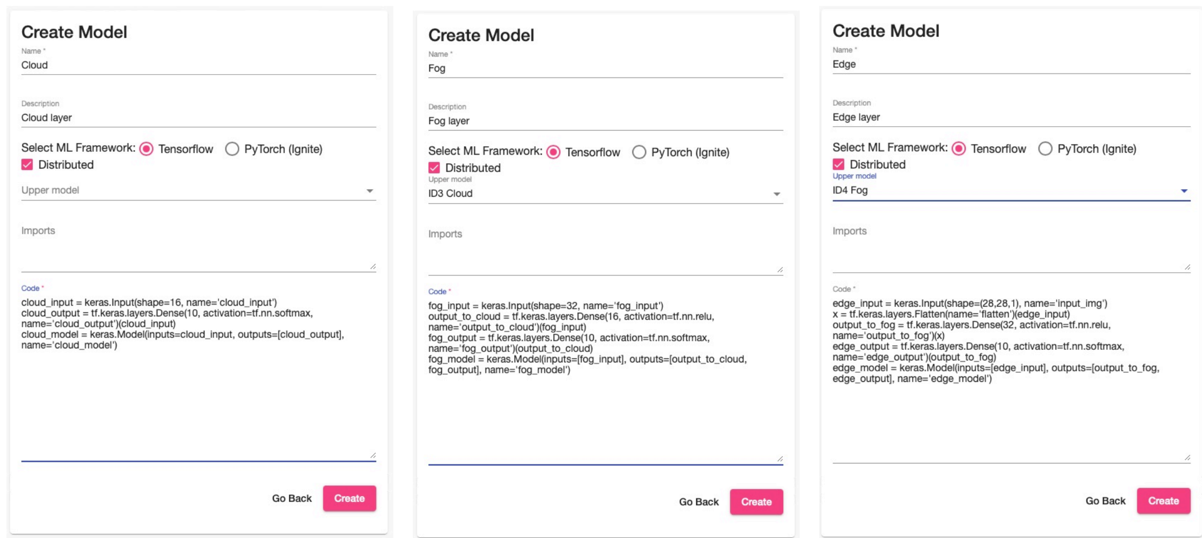
Listing 2: Ejemplo de despliegue de un modelo utilizando NewDeploy

5 Evaluación

Para evaluar el comportamiento de nuestro sistema, simulamos un dispositivo IoT que publica datos en el tópico de entrada de la capa Edge. Los mensajes se envían a razón de un mensaje por segundo, reduciendo el tiempo entre mensajes cada 100 mensajes hasta alcanzar 400 mensajes. Nuestra intención es observar el comportamiento de cada plataforma bajo una carga variable, similar a la que podríamos observar en situaciones reales.

5.1. Entorno de pruebas

Nuestro modelo fue entrenado utilizando Kafka-ML con el conjunto de datos de entrenamiento MNIST (Figura 5). El modelo está distribuido en tres capas siguiendo la arquitectura de BranchyNet [53]: Cloud, Fog y Edge. La capa Cloud (Figura 5a) recibe una entrada con una forma de 16 y tiene una capa de salida con 10 nodos, utilizando una activación softmax. La capa Fog (Figura 5b) recibe una entrada con una forma de 32, la procesa a través de una capa densa con una función de activación de unidad lineal rectificada (ReLU) y la envía a la capa Cloud. También tiene una capa de salida con 10 nodos que utiliza una activación softmax. Finalmente, la capa de Edge (Figura 5c) recibe imágenes de entrada con una forma de 28x28x1 y las procesa a través de una capa densa con una función de activación ReLU. La salida se pasa luego a la capa Fog. Esta también tiene una capa de salida con 10 nodos con una activación softmax. Los modelos son entrenados utilizando el framework TensorFlow, y cada nivel se define como un modelo separado. La versión de TensorFlow utilizada para el entrenamiento es la 2.7.0, mientras que la versión utilizada para la inferencia es la 2.11.0. A diferencia de la mayoría de los modelos de clasificación de imágenes que utilizan redes neuronales convolucionales, nuestro modelo está compuesto por capas densas. La razón de esta decisión fue aleatorizar la salida del modelo, ya que las predicciones alcanzaban el 100 % de precisión en la primera capa utilizando redes convolucionales.



(a) Cloud (b) Fog (c) Edge

Figura 5: Definición de los modelos en Kafka-ML

Cada capa fue desplegada en un cluster de Kubernetes distinto con las siguientes especificaciones:

- **Cluster Cloud *on premise***: Compuesto por 7 nodos de última generación. Cada máquina tiene un procesador Intel(R) Xeon(R) Gold 6230R con dos GPUs NVIDIA(R) Tesla(R) V100 y 384 GB de RAM. Estos nodos ejecutan Kubernetes v1.21.6 y Docker 20.10.8 sobre Ubuntu 20.04.3 LTS. Se desplegó un nodo master de Kubernetes en una máquina, mientras que las seis restantes son slaves de Kubernetes. Ejecutamos Kafka usando el Helm chart `bitnami/kafka` versión 21.3.1, con dos brokers y la persistencia desactivada. Este Kafka se encarga del tópico de entrada de esta capa. El número de particiones para el topico de entrada es 1.

- **Cluster Fog**: Un único nodo compuesto por un procesador Intel(R) Core(TM) i9-10900K y 64 GB de RAM. Este nodo ejecuta K3s v1.25.7+k3s1 sobre Ubuntu 21.04, con Traefik desactivado. Desplegamos Kafka usando el Helm chart `bitnami/kafka` versión 21.3.1, con un broker y la persistencia desactivada. Este Kafka se encarga del tópico de entrada de esta capa. El número de particiones para el tópico de entrada es 1.

- **Cluster Edge**: 6 Raspberry Pi 3B+, con 1 GB de RAM y una tarjeta SD Samsung EVO Plus de 64 GB cada uno. Cada Raspberry tiene su propia fuente de alimentación micro

USB Raspberry de 12,5 W. Estos nodos ejecutan K3s v1.25.7+k3s1, con Traefik desactivado, sobre Raspberry Pi OS lite (64-bit) versión 2023-02-21. Ejecutamos Kafka usando el Helm chart `bitnami/kafka` versión 21.3.1, con un broker y la persistencia desactivada. Tuvimos que actualizar la imagen de Zookeper a `3.8.1-debian-11-r8`, ya que es la primera versión 3.8 con soporte para ARM. Este Kafka se encarga del tópico de entrada de esta capa, el tópico de salida del Cloud y los *early exits* de las capas Fog y Edge. El número de particiones para estos tópicos es 1.

Sobre las plataformas *serverless*, OpenFaaS fue desplegado utilizando el Helm chart, versión 12.0.2 con 4 réplicas del componente Gateway en las capas Edge y Cloud, mientras que en el Fog solo desplegamos una. Fission fue desplegado utilizando el Helm chart en las tres capas, en su versión v1.18.0 con InfluxDB activado, persistencia desactivada y "deployAsDaemonSet" en el componente Router.

En ambas plataformas, establecimos el número de mínimo de réplicas de la función inferencia en 1, y el máximo en 20. A esta función se le asigno 1000m de CPU y 1024MB de memoria en el Fog y Cloud. En el Edge, se configuró con 500m de CPU y 512MB de memoria. De forma similar, la función *funnel* de desplegó con un mínimo de 1 réplica y un máximo de 10 en todas las capas.

5.2. Resultados obtenidos

La Figura 6 ilustra los resultados obtenidos, donde el tiempo de respuesta máximo para las tareas de inferencia es de alrededor de 926 segundos, registrado por la capa Edge de OpenFaaS. Por otro lado, el tiempo de respuesta mínimo fue de alrededor de 3 segundos, también registrado por la capa Edge de OpenFaaS. El tiempo de respuesta medio para todos los tópicos varía entre 430 segundos y 439 segundos, siendo la capa Cloud de Fission la que obtiene el tiempo de respuesta medio más bajo y la capa Fog de OpenFaaS la que obtiene el tiempo de respuesta medio más alto. La mediana, que es menos sensible a los valores atípicos, varía entre de 417 segundos y 422 segundos en todos los tópicos. La desviación estándar, que mide la cantidad de variabilidad en los datos, se sitúa entre 270 segundos y 274 segundos. Valores altos de desviación estándar indican que hay una cantidad significativa de variabilidad en las mediciones de latencia.

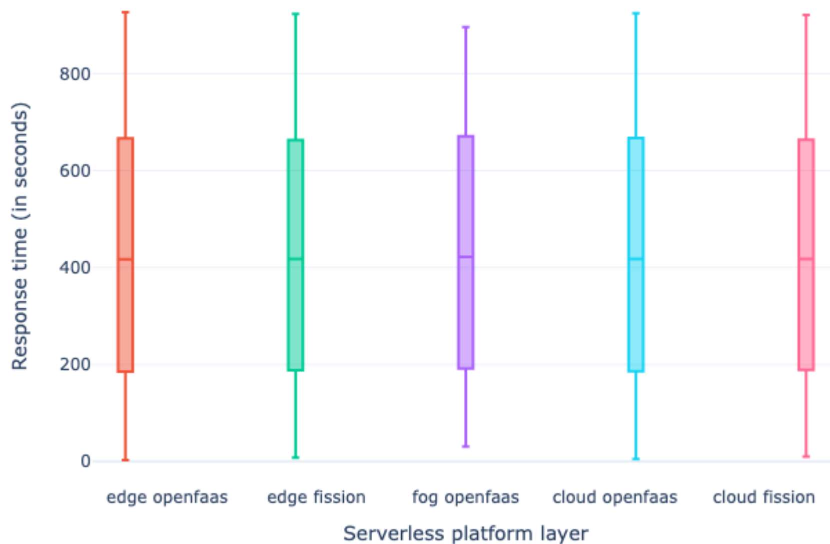


Figura 6: Diagrama de caja de los tiempos de respuesta por cada plataforma y capa

A pesar de los altos tiempos de respuesta que experimentamos, OpenFaaS no escaló ninguna de las funciones en ninguna de las capas. Sin embargo, Fission escaló la función de inferencia de una réplica a dos en nuestra capa Edge. A pesar de la diferencia en el número de réplicas, no medimos diferencias significativas entre los tiempos de respuesta de ambas plataformas. De hecho, OpenFaaS obtuvo tiempos de respuesta medios más bajos en comparación con Fission.

5.3. Interpretación de los resultados

Después de analizar los resultados obtenidos, así como las métricas de rendimiento recopiladas por el Prometheus de OpenFaaS, descubrimos que nuestro problema de escalado de funciones se debía a un bajo número de peticiones por segundo. Nuestro sistema no era capaz de procesar datos con la misma velocidad con la que se generaban, dando lugar a esos tiempos de respuesta tan altos. Concluimos que esto se debió principalmente a los limitados recursos disponibles en nuestros nodos Edge, concretamente a la cantidad de memoria y a la velocidad del disco de estos nodos. Estos mantienen al rededor de 100-150 megabytes de memoria disponibles sin aplicaciones desplegadas en K3s. Al desplegar el sistema, la cantidad de memoria

disponible se reducía, obligando al kernel intercambiar páginas con el disco para reducir la presión de memoria y, consecuentemente, ralentizando significativamente nuestro sistema.

Nuestro cluster Edge es capaz de ejecutar modelos básicos de DDNN como éste, pero sospechamos que los componentes instalados junto con las plataformas FaaS que evaluamos pudieron haber sobrecargado nuestro cluster. Concretamente, OpenFaaS requiere al menos un Gateway, un AlertManager, Prometheus y una cola NATS. Fission, por otro lado, requiere del gestor de paquetes, un Router por nodo del cluster y un constructor por entorno. Además, cada función en Fission incluye un contenedor adicional (el mencionado fetcher), aumentando los recursos necesarios para ejecutar cada función.

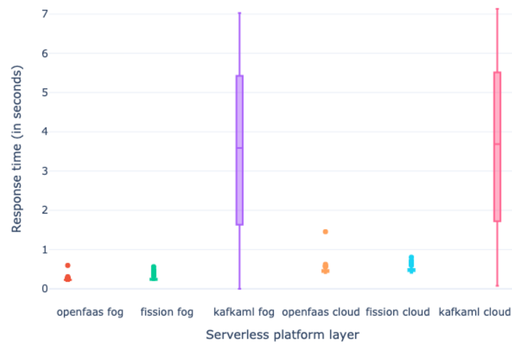
6
Evaluación sin Edge

Tras el análisis de los resultados obtenidos en la Sección 5, optamos por modificar nuestro sistema. Dado que la capa Edge supone un cuello de botella, decidimos fusionar las capas Edge y Fog. También desplegamos nuestro modelo utilizando Kafka-ML con el objetivo de comparar el rendimiento entre ambas implementaciones. Estas difieren en varios aspectos: Kafka-ML utiliza un formato binario para la comunicación intra-capa, mientras que nuestra aplicación utiliza JSON. Además, Kafka-ML utiliza una arquitectura monolítica donde el consumidor de Kafka, el productor y los componentes de inferencia residen en el mismo contenedor, diferente de la arquitectura de nuestra aplicación. Por último, la inferencia de Kafka-ML no escala el número de réplicas automáticamente.

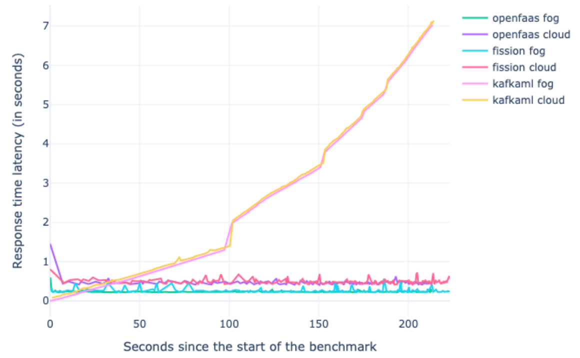
6.1. Resultados obtenidos

La Figura 7 ilustra los resultados obtenidos para un solo cliente. En la Figura 7a observamos que las plataformas OpenFaaS y Fission mantuvieron latencias similares. Los valores medianos para ambas plataformas rondan los 230ms para Fog y 470ms para Cloud. OpenFaaS exhibió los tiempos de respuesta medios, mediana y mínimos más bajos en ambas capas. La Figura 7b presenta los datos, ilustrando las latencias obtenidas a lo largo del tiempo por cada plataforma y capa. Las latencias obtenidas por OpenFaaS se mantuvieron en su mayoría consistentes durante el análisis, mientras que Fission experimentó picos de latencia más grandes. Es importante señalar que ninguna de las dos plataformas escaló el número de réplicas durante la prueba. En comparación con Kafka-ML, OpenFaaS acabó antes (213s), seguido de Kafka-ML (214s) y, por último, Fission (223s).

Al aumentar el número de clientes a tres, observamos diferentes comportamientos entre OpenFaaS y Fission (Figura 8). OpenFaaS escaló la función `inference` en el Fog hasta cinco réplicas, mientras que Fission solo instanció dos réplicas. Sorprendentemente, Fission com-

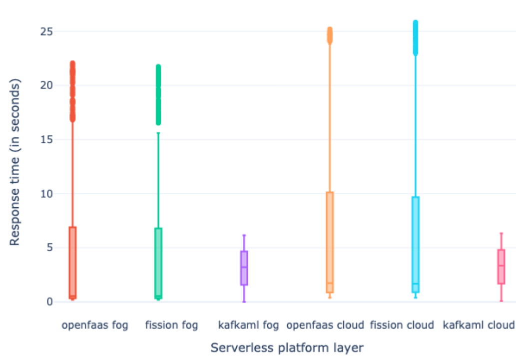


(a) Diagrama de caja de las latencias obtenidas

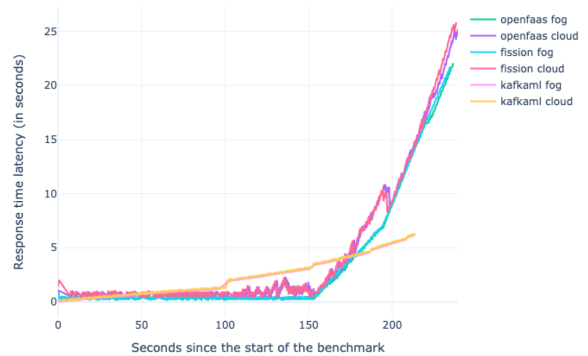


(b) Latencias con respecto el paso del tiempo

Figura 7: Resultados obtenidos con un solo cliente

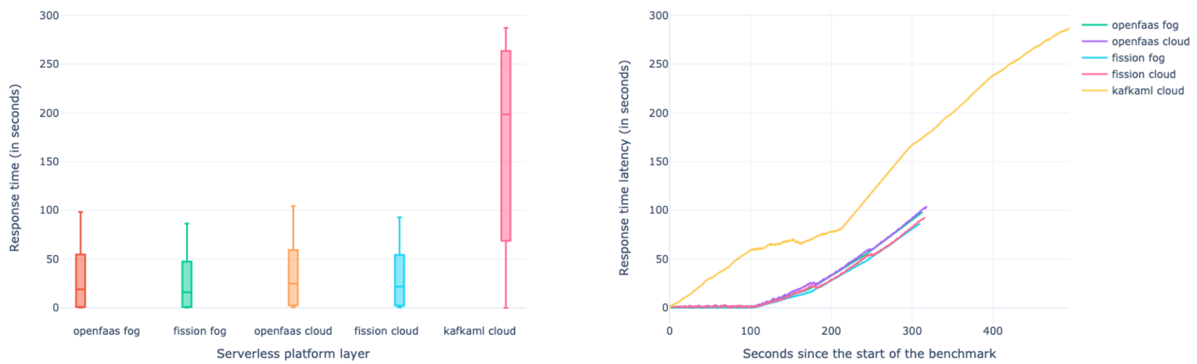


(a) Diagrama de caja de las latencias obtenidas



(b) Latencias con respecto el paso del tiempo

Figura 8: Resultados obtenidos con tres clientes



(a) Diagrama de caja de las latencias obtenidas

(b) Latencias con respecto el paso del tiempo

Figura 9: Resultados obtenidos con cinco clientes

pletó la prueba antes que OpenFaaS, a pesar de instanciar menos de la mitad del número de réplicas. La Figura 8a ilustra los resultados, indicando que ambas plataformas tuvieron tiempos de respuesta similares. La Figura 8b presenta los datos, donde podemos observar que nuestro conector de Kafka alcanzó su límite en la tercera etapa, alrededor de los 150 segundos. Por último, en comparación con Fission (238s) y OpenFaaS (239s), Kafka-ML tuvo el tiempo de inferencia más corto (213s).

Finalmente, aumentamos el número de clientes a cinco, lo que provocó que el conector alcanzara su límite a los 100 segundos. OpenFaaS y Fission exhibieron un comportamiento de escalado similar al anterior, con OpenFaaS instanciando cinco réplicas y Fission dos. Una vez más, Fission completó la prueba más rápido que OpenFaaS. La Figura 9b representa los datos para esta etapa, mientras que la Figura 9a presenta los resultados. Fission terminó el más rápido (315s), seguido de OpenFaaS (318s) y por último Kafka-ML (494s).

6.2. Interpretación de los resultados

En la primera ronda, con solo un dispositivo simulado, observamos resultados similares a los obtenidos por Mohanty et al. [54]. Fission presenta muchos valores atípicos en latencia, posiblemente consecuencia del componente Router. Esto hizo que la plataforma fuese más lenta que Kafka-ML y OpenFaaS. Durante esta fase, también podemos ver cómo OpenFaaS fue

un segundo más rápido que Kafka-ML. Atribuimos este resultado a pequeñas ganancias en paralelismo: mientras que nuestra función de inferencia estaba realizando el cálculo, nuestro conector de Kafka ya se encontraba listo para extraer el siguiente valor del tópico.

En la segunda ronda, con tres dispositivos simulados, podemos ver que tanto OpenFaaS como Fission fueron 10 % más lentos que Kafka-ML. Creemos que esto es consecuencia de los *cold starts* de ambas plataformas, ya que tanto OpenFaaS como Fission instanciaron varias réplicas. Además, sospechamos que las transferencias de datos producidas entre el conector y las funciones ralentizaron la inferencia.

Por último, al aumentar el número de clientes a cinco, tanto OpenFaaS como Fission fueron capaces de procesar los datos más rápido que Kafka-ML. Los resultados indican un 55 % de mejora en el rendimiento sobre la implementación de DDNN de Kafka-ML. Destacamos también la eficiencia de Fission, ya que mantuvo el número de réplicas bajo y más estable, permitiendo a la plataforma acabar la prueba antes.

Nuestros experimentos también revelaron que OpenFaaS exhibe un comportamiento de escalado más agresivo en comparación con Fission. OpenFaaS creaba y eliminaba muchas instancias de nuestras funciones, incluso si estas se encontraban todavía en la fase de "ContainerCreating"[2]. Por el contrario, Fission solo escalaba las funciones cuando el sistema no era capaz de soportar la carga, y eliminaba las instancias cuando estas estaban inactivas por un tiempo.

[2]"ContainerCreating"significa que Kubernetes esta iniciando el contenedor, concretamente descargando la imagen e iniciando el sistema de ficheros

7 Conclusiones y Líneas Futuras

7.1. Conclusiones

En este trabajo presentamos una visión general de las tres plataformas *serverless* de código abierto más populares, con énfasis en el proceso de instalación, la experiencia de desarrollo y cualidades únicas de cada plataforma. Además, exploramos el potencial de la computación *serverless* al construir una aplicación IoT *serverless* de baja latencia.

OpenFaaS y Fission son las plataformas más completas y fáciles con las que desarrollar, ya que proporcionan abstracciones cruciales que hacen que la experiencia de desarrollo sea muy sencilla. En el caso de OpenWhisk, concluimos que la plataforma es inestable y no funciona correctamente. OpenFaaS es la plataforma más amigable de todas. Ofrece plantillas para crear funciones, una gran colección de lenguajes soportados y una excelente experiencia de desarrollo gracias a su CLI, aunque la compilación en otras arquitecturas puede ser tediosa. Sin embargo, encontramos su modelo de negocio preocupante, ya que OpenFaaS restringe el acceso a elementos esenciales a su versión pro. Esto puede dar lugar a la indeseada dependencia con el proveedor, comúnmente observada con otras soluciones propietarias.

Fission es la plataforma más flexible, ya que adopta por completo las características de Kubernetes, como los dispositivos y volúmenes. Esto deja a la plataforma limitada a Kubernetes, pero dado lo común que el orquestador es, difícilmente podemos verlo como una desventaja. Fission también presenta una gran experiencia de desarrollo, especialmente al compilar el código fuente de nuestra función. Sin embargo, Fission fue la plataforma más pesada, ya que creó más pods de control y contenedores que OpenFaaS.

Concluimos que *serverless* simplifica la experiencia del desarrollador, pero requiere más

recursos disponibles que la computación tradicional. Por lo tanto, no consideramos OpenFaaS CE o Fission opciones adecuadas para entornos con recursos limitados (es decir, Edge). Sin embargo, para entornos no limitados como Fog y Cloud, *serverless* podría considerarse una alternativa a los despliegues monolíticos tradicionales.

Como resultado de este trabajo Fin de Grado se ha desarrollado el artículo "Functions as a Service for Distributed Deep Neural Network inference over the Cloud-to-Things continuum", que actualmente se encuentra en revisión en el congreso 2023 International Conference on Service Oriented Computing.

7.2. Líneas Futuras

Como líneas futuras, planeamos explorar las posibilidades de *serverless* en otras situaciones. Una posible línea de investigación sería el procesamiento de flujos sin servidor a través del *Cloud-to-Things continuum*, siguiendo la arquitectura descrita en este trabajo. Aunque existen trabajos previos en este área [55], estos se limitan a la infraestructura AWS Lambda.

Referencias

[1] Cristian Martín et al. «Kafka-ML: Connecting the data stream with ML/AI frameworks». En: *Future Generation Computer Systems* 126 (2022), págs. 15-33. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2021.07.037. URL: https://www.sciencedirect.com/science/article/pii/S0167739X21002995.

[2] *Website de Docker*. Disponible online: https://www.docker.com/. (accedido el 28 de Enero de 2023).

[3] *Website de Kubernetes*. Disponible online: https://kubernetes.io. (accedido el 28 de Enero de 2023).

[4] *Website de la Cloud Native Computing Foundation*. Disponible online: https://www.cncf.io/. (accedido el 28 de Enero de 2023).

[5] *Cloud Native Computing Foundation accepts Kubernetes as first hosted project; Technical Oversight Committee elected*. Disponible online: https://www.cncf.io/announcements/2016/03/10/cloud-native-computing-foundation-accepts-kubernetes-as-first-hosted-project-technical-oversight-committee-elected/. (accedido el 28 de Enero de 2023).

[6] *Certified Kubernetes Software Conformance*. Disponible online: https://www.cncf.io/certification/software-conformance/. (accedido el 28 de Enero de 2023).

[7] *Website de K3s*. Disponible online: https://k3s.io/. (accedido el 28 de Enero de 2023).

[8] Gregorio Robles y Jesús M. González-Barahona. «A Comprehensive Study of Software Forks: Dates, Reasons and Outcomes». En: *Open Source Systems: Long-Term Sustainability*. Ed. por Imed Hammouda et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, págs. 1-14. ISBN: 978-3-642-33442-9.

[9] *Website de Helm*. Disponible online: https://helm.sh/. (accedido el 28 de Enero de 2023).

[10] *Openfaas/faas: Openfaas - serverless functions made simple*. Disponible online: https://github.com/openfaas/faas. (accessed on 24 January 2023).

[11] *Website de la Cloud Native Computing Foundation Landscape*. Disponible online: https://landscape.cncf.io/. (accedido el 28 de Enero de 2023).

[12] *Fission/fission: Fast and simple serverless functions for Kubernetes*. Disponible online: https://github.com/fission/fission. (accessed on 24 January 2023).

[13] *Apache/openwhisk: Apache OpenWhisk is an open source serverless cloud platform*. Disponible online: https://github.com/apache/openwhisk. (accessed on 24 January 2023).

[14] *IBM Cloud Functions*. Disponible online: https://www.ibm.com/es-es/cloud/functions. (accessed on 24 January 2023).

[15] *Website de Python*. Disponible online: https://www.python.org/. (accedido el 28 de Enero de 2023).

[16] *Website de Tensorflow*. Disponible online: https://www.tensorflow.org/. (accedido el 28 de Enero de 2023).

[17] *Website de Jupyter*. Disponible online: https://jupyterbook.org/en/stable/intro.html. (accedido el 28 de Enero de 2023).

[18] *Website de TypeScript*. Disponible online: https://www.typescriptlang.org/. (accedido el 28 de Enero de 2023).

[19] *Website de Deno*. Disponible online: https://deno.land/. (accessed on 24 January 2023).

[20] *Website de NodeJS*. Disponible online: https://nodejs.org/en. (accedido el 28 de Enero de 2023).

[21] *Website de Kafka*. Disponible online: https://kafka.apache.org/. (accedido el 28 de Enero de 2023).

[22] *Website de Raspberry Pi*. Disponible online: https://www.raspberrypi.com/. (accedido el 28 de Enero de 2023).

[23] *Website de Visual Studio Code*. Disponible online: https://code.visualstudio.com/. (accedido el 28 de Enero de 2023).

[24] *Website de Git*. Disponible online: https://git-scm.com. (accedido el 28 de Enero de 2023).

[25] *Website de GitHub*. Disponible online: https://github.com/. (accedido el 28 de Enero de 2023).

[26] *Microsoft acquires GitHub*. Disponible online: https://news.microsoft.com/announcement/microsoft-acquires-github/. (accedido el 28 de Enero de 2023).

[27] Gojko Adzic y Robert Chatley. «Serverless Computing: Economic and Architectural Impact». En: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, págs. 884-889. ISBN: 9781450351058. DOI: 10.1145/3106237.3117767. URL: https://doi.org/10.1145/3106237.3117767.

[28] Theodore Lynn et al. *The Cloud-to-Thing Continuum Opportunities and Challenges in Cloud, Fog and Edge Computing: Opportunities and Challenges in Cloud, Fog and Edge Computing*. Ene. de 2020. ISBN: 978-3-030-41109-1. DOI: 10.1007/978-3-030-41110-7.

[29] Ioana Baldini et al. «Serverless computing: Current trends and open problems». En: *Research advances in cloud computing* (2017), págs. 1-20.

[30] Desire Harauzek. «Cloud Computing : Challenges of cloud computing from business users perspective - vendor lock-in». Tesis doct. 2022. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-115089.

[31] *OpenFaaS docs: Legacy scaling for the Community Edition (CE)*. Disponible online: https://docs.openfaas.com/architecture/autoscaling/#legacy-scaling-for-the-community-edition-ce. (accessed on 28 January 2023).

[32] *OpenFaaS docs: CLI Templates*. Disponible online: https://docs.openfaas.com/cli/templates/#templates. (accessed on 28 January 2023).

[33] *OpenFaaS docs: Deployment*. Disponible online: https://docs.openfaas.com/deployment/. (accessed on 24 January 2023).

[34] *OpenFaaS docs: Deployment*. Disponible online: https://docs.openfaas.com/deployment/. (accessed on 24 January 2023).

[35] *OpenFaaS docs*. Disponible online: https://docs.openfaas.com/. (accessed on 24 January 2023).

[36] *OpenFaaS docs: Your first OpenFaaS Function with Python*. Disponible online: https://docs.openfaas.com/tutorials/first-python-function/. (accessed on 24 January 2023).

[37] *Fission docs: Concepts*. Disponible online: https://fission.io/docs/concepts/. (accessed on 24 January 2023.

[38] *Kubernetes docs: Horizontal Pod Autoscaling*. Disponible online: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. (accessed on 28 January 2023).

[39] *Fission docs: Executor*. Disponible online: https://fission.io/docs/architecture/executor/. (accessed on 24 January 2023).

[40] *Kubernetes docs: Understanding Kubernetes Objects*. Disponible online: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/. (accessed on 28 January 2023).

[41] *Fission docs: Installing Fission*. Disponible online: https://fission.io/docs/installation/. (accessed on 24 January 2023).

[42] *Fission docs*. Disponible online: https://fission.io/docs/. (accessed on 24 January 2023).

[43] *Kubernetes docs: Custom resources*. Disponible online: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/. (accessed on 28 January 2023).

[44] *apache/openwhisk-deploy-kube/docs/k8s-diy.md*. Disponible online: https://github.com/apache/openwhisk-deploy-kube/blob/e202f469181716ba15e0676245680057d294d951/README.md. (accessed on 24 January 2023).

[45] *openwhisk-deploy-kube/docs/troubleshooting*. Disponible online: https://github.com/apache/openwhisk-deploy-kube/blob/e202f469181716ba15e0676245680057d294d951/docs/troubleshooting.md. (accessed on 24 January 2023).

[46] Alejandro Carnero et al. «Managing and Deploying Distributed and Deep Neural Models Through Kafka-ML in the Cloud-to-Things Continuum». En: *IEEE Access* 9 (2021), págs. 125478-125495. DOI: 10.1109/ACCESS.2021.3110291.

[47] Daniel R. Torres et al. «An open source framework based on Kafka-ML for Distributed DNN inference over the Cloud-to-Things continuum». En: *Journal of Systems Architecture* 118 (2021), pág. 102214. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2021.102214. URL: https://www.sciencedirect.com/science/article/pii/S138376212100151X.

[48] *openfaas/python-flask-template: HTTP and Flask-based OpenFaaS templates for Python 3*. Disponible online: https://github.com/openfaas/python-flask-template. (accessed on 29 January 2023).

[49] *OpenFaaS Issue Tracker: Research: show GPU attached to a function*. Disponible online: https://github.com/openfaas/faas/issues/639. (accessed on 28 January 2023).

[50] *OpenFaaS Issue Tracker: Question: How do I publish images when buildkit requires a self-signed certificate?* Disponible online: https://github.com/openfaas/faas-cli/issues/954. (accessed on 28 January 2023).

[51] *fission/environments: Fission: Python Environment*. Disponible online: https://github.com/fission/environments/tree/b0c38112f9a222a8101450238e6b2b31a9e857ce/python. (accessed on 10 April 2023).

[52] *Python Packages in OpenWhisk*. Disponible online: https://jamesthom.as/2017/04/python-packages-in-openwhisk/. (accessed on 24 January 2023).

[53] Surat Teerapittayanon, Bradley McDanel y H.T. Kung. «BranchyNet: Fast inference via early exiting from deep neural networks». En: *2016 23rd International Conference on Pattern Recognition (ICPR)*. 2016, págs. 2464-2469. DOI: 10.1109/ICPR.2016.7900006.

[54] Sunil Kumar Mohanty, Gopika Premsankar y Mario di Francesco. «An Evaluation of Open Source Serverless Computing Frameworks». En: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2018, págs. 115-120. DOI: 10.1109/CloudCom2018.2018.00033.

[55] Sebastian Werner, Richard Girke y Jörn Kuhlenkamp. «An Evaluation of Serverless Data Processing Frameworks». En: *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*. WoSC'20. Delft, Netherlands: Association for Computing Machinery, 2021, págs. 19-24. ISBN: 9781450382045.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA