Check for updates

Received: 19 July 2023 | Revised: 19 December 2023 | Accepted: 12 January 2024

DOI: 10.1002/spe.3318

RESEARCH ARTICLE

WILEY

Functions as a service for distributed deep neural network inference over the cloud-to-things continuum

Altair Bueno | **Bartolomé Rubio** | **Cristian Martín** | **Manuel Díaz**

ITIS Software, University of Málaga, Málaga, Spain

Correspondence
Altair Bueno, ITIS Software, University of Málaga, Ada Byron Research Building, C/Arquitecto Francisco Peñalosa 18, 29071, Málaga, Spain.
Email: altair@uma.es

Funding information
Ministerio de Ciencia, Innovación y Universidades

Abstract

The use of serverless computing has been gaining popularity in recent years as an alternative to traditional Cloud computing. We explore the usability and potential development benefits of three popular open-source serverless platforms in the context of IoT: OpenFaaS, Fission, and OpenWhisk. To address this we discuss our experience developing a serverless and low-latency Distributed Deep Neural Network (DDNN) application. Our findings indicate that these serverless platforms require significant resources to operate and are not ideal for constrained devices. In addition, we archived a 55% improvement compared to Kafka-ML's performance under load, a framework without dynamic scaling support, demonstrating the potential of serverless computing for low-latency applications.

KEYWORDS

cloud-to-things continuum, DDNN inference, Fission, Kafka-ML, OpenFaaS, OpenWhisk, serverless computing

1 | INTRODUCTION

In recent years, Functions as a Service (FaaS) has proven to be a valid alternative approach to traditional Cloud computing. Its stateless nature allows them to scale horizontally on-demand by reacting to events, which translates to multiple instances running under heavy load, or no instances at all. This automatic scaling is cost-effective,[1] allowing developers to focus on solving problems instead of worrying about the underlying infrastructure.

IoT applications generate massive amounts of data that need to be handled efficiently and flexibly. It is essential to have Cloud and Fog/Edge computing solutions that can react to these unpredictable flows of data. We consider FaaS as a promising solution for simplifying the development and deployment of such applications over the Cloud-to-Things continuum.[2]

One of these applications that could benefit from FaaS flexibility is Deep Neural Networks (DNN), which have witnessed a surge in popularity, finding applications across diverse domains. In the realm of IoT, DNNs have proven to be highly successful, proving to be useful in areas such as soft sensors[3] and defect detection.[4] DNN often require a large number of computational resources and specialized hardware (e.g., GPUs), which may not satisfy the requirements of resource-constrained systems. Distributing the DNN (DDNN) over the Cloud-to-Things continuum allows responses to be

Abbreviations: DDNN, distributed deep neural network; DNN, deep neural network; FaaS, function as a service; IoT, internet of things.

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.
© 2024 The Authors. *Software: Practice and Experience* published by John Wiley & Sons Ltd.

generated as close as possible to where the information is produced, minimising round trip times, and meeting the requirements of embedded devices. We had previous experience in deploying DDNNs in the Cloud-to-Things continuum[5,6] through our Kafka-ML framework,[7] however, we had a number of limitations when the load dynamically increased, which we could not handle with the approach adopted. By leveraging FaaS, we expect similar levels of response times as those archived with traditional DDNNs, but with automatic horizontal scaling when required.

Although major Cloud providers offer FaaS solutions, these tend to be closed-source and can result in vendor lock-in.[8,9] The purpose of this paper is to provide a general overview of the three most popular[*] open-source FaaS platforms, with an emphasis on their potential development benefits in developing a low-latency IoT application. In particular, we consider a DDNN deployment over different layers of the Cloud-to-Things continuum.

Therefore, the main contributions of this article are:

1. An analysis of three open-source FaaS frameworks with a focus on the potential development benefits.
2. The development of an open-source DDNN inference system based on FaaS for `x86_64` and `arm64` machines.
3. A performance evaluation of the system with DDNN inference across the Cloud-to-Things continuum.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 describes our DDNN serverless architecture. Sections 4,5 and 6 provide an introduction, overview of the development experience, and our findings during the development of our DDNN for OpenFaaS, Fission, and OpenWhisk, respectively. Section 7 shows the results obtained from our validation tests. In Section 8 we revaluated our application without the Edge layer. Finally, we lay out our conclusions and future work in Section 9.

2 | RELATED WORK

Mohanty et al.[11] analyzed the status of open-source serverless computing frameworks, particularly a feature comparison between OpenFaaS, Fission, OpenWhisk, and Kubeless. The authors focused on evaluating the performance of these platforms on Google Kubernetes Engine and concluded that OpenFaaS had the most flexible architecture, while Kubeless had the most consistent performance across different scenarios. A similar performance analysis was conducted by Balla et al.,[12] where they showcase the performance differences of the Python3, Node.js and Golang runtimes of OpenFaaS, Kubeless, Fission, and Knative platforms, as well as the influence of the different auto-scaling algorithms over the function runtimes. Their experiments aimed at comparing IO-bounded and compute-bounded functions on each platform. In this work, we evaluate these platforms on a low-latency scenario and also provide a clear overview of each platform's features and potential benefits to the development workflow.

A similar evaluation focused on resource-constrained, edge computing environment was performed by Palade et al.[13] They evaluated four open-source serverless frameworks, namely Kubeless, Apache OpenWhisk, Knative, and OpenFaaS. They concluded that Kubeless outperforms the other frameworks across the proposed scenarios in terms of response time and throughput, and Apache OpenWhisk had the worst performance of all. We expanded their work by assessing these platforms on a constrained Edge layer, composed of `arm64` Raspberry Pi machines.

Jindal et al.[14] focused on scheduling FaaS on heterogeneous functions, platforms, and hardware. They introduced an extension called Function Delivery Network, which aims to simplify the development and deployment of heterogeneous FaaS on OpenFaaS, Google Cloud Functions, and OpenWhisk. The authors concluded that by bringing data closer to the target platform, significant reductions in data access latency could be achieved. Our research also combines heterogeneous hardware and functions but not FaaS platforms. While we acknowledge the valuable features offered by each platform, we believe that managing different functions across multiple platforms adds complexity that outweighs the benefits of utilizing FaaS.

Related to DNN, execution has been attempted before on platforms such as AWS Lambda, notably Park et al.,[15] where the authors proposed a system as a service using various fully-managed cloud solutions. The system was released as open-source software to help DNN application developers to build an optimal serving environment. Contrary to this research, we use only open-source software that can be self-hosted and we distribute the DNN across the Cloud-to-Things continuum.

[*]Popularity rank based on the current number of stargazers on their GitHub repository.[10]

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3318 by Chua-Consorcio De Bibliotecas, Wiley Online Library on [14/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Cirrus[16] is another serverless framework that utilizes machine learning for task automation. This framework aims to provide more efficient workflows by leveraging serverless infrastructures. The study found that Cirrus was faster and more resource-efficient than traditional ML frameworks. However, this research does not consider DDNN over the Cloud-to-Things continuum and is not based on open-source software.

Gillis[17] provides a serverless inference system that dynamically partitions DNN across multiple serverless functions for fast inference. Although dynamic partitioning is promising, unlike our work, Gillis does not consider open-source serverless platforms and does not take into account early exits and the Cloud-to-Things continuum for distributing the functions.

On self-hosted infrastructure, frameworks like Kafka-ML[7] have made it easier to deploy DDNN over multiple Kubernetes clusters. Kafka-ML is a novel and open-source framework for managing ML/AI pipelines, which leverages Kafka streams to perform DDNN inference, similarly to our research. Additionally, Kafka-ML shares several key design decisions and architectural aspects with our work, making it a suitable point of reference for our experiments. To the best of our knowledge it is the only open source framework that is capable of handling deep and distributed neural networks over data streams. It is true that other platforms such as All-You-Can-Inference[15] and Cirrus[16] have similar objectives, but in the context of this work, which is to study the feasibility of using lambda functions on these models with continuous data, we have considered it appropriate to use Kafka-ML as a baseline model, since it is the only one that can directly evaluate our objective. However, compared to our system, the deployment is still limited to a fixed number of pods, which means that scaling the system requires manual interaction with Kafka-ML. We will consider the Kafka-ML framework to assess the performance of our architecture against a monolithic framework with no dynamic scalability.

3 | ARCHITECTURE FOR DDNN WITH FUNCTIONS AS A SERVICE

Our architecture is composed of DDNN layers that run on separate serverless platforms over the Cloud-to-Things continuum (Edge, Fog, Cloud). Following Torres et al.[6] design, each layer requires two outputs: one is reserved for early exits (intermediary prediction in each layer of the Cloud-to-Things continuum to reduce the prediction latency) of the DDNN layer, while the other one is used to bubble up the inference to the next layer (if there is one). When the prediction accuracy of a layer is sufficiently high, the prediction is made by the early exit saving in communications, otherwise, the prediction is rising to the next layer. In this work, we address the limitations of this architecture with the provision of FaaS. Figure 1 illustrates our architecture's design for DDNN inference with FaaS.

All layers communicate through Apache Kafka, the open-source distributed event streaming platform. Kafka operates as a distributed messaging system, based on the publish/subscribe model to efficiently dispatch and consume large amounts of data with minimal latency. Unlike other message queues, Kafka's publish/subscribe systems allow multiple consumers to receive each message within a given topic. Additionally, in contrast to traditional message queue systems where messages are often deleted after consumption, Kafka's distributed log eliminates the need for external data stores to persist the data. This feature is valuable in data operations, as streams can be repurposed for lots of tasks involving FaaS, such as async batch processing and request buffering.

FaaS are designed to produce only one output per function invocation. One way to solve this limitation is to connect our inference functions directly to our Kafka brokers (Kafka's main architecture peers). The function then writes the results directly to Kafka, depending on whether the inference was an early exit or the input for the next layer. This approach poses some scaling issues as each function instance would maintain its own connection pool, potentially overloading the brokers if we instantiate many copies of our function at once.

We settled on a different architecture for our implementation, where we have two functions with different purposes. The first function, called `funnel`, is responsible for maintaining the connection with Kafka and writing incoming requests to a specific Kafka topic. The second function, called `inference`, performs the inference over the streaming and invokes either the output funnel or the next layer's input funnel with the results.

In our experimentation, we did not observe significant performance impacts or service degradation caused by direct communication between the sink function and Kafka. In fact, latency improved under extreme load as the funnel function could easily scale up independently when needed, as we will later see on Section 8. Furthermore, we wanted to showcase the extensibility of functions thought composition, which allows FaaS to archive complex workflows from simpler specialized functions.

For connecting our functions to the Kafka stream, we tried using each serverless platform's Kafka connector. In the context of these platforms, a connector is essentially an application that awaits events and then proxies the payload

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3318 by Chua-Consorcio De Bibliotecas, Wiley Online Library on [14/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

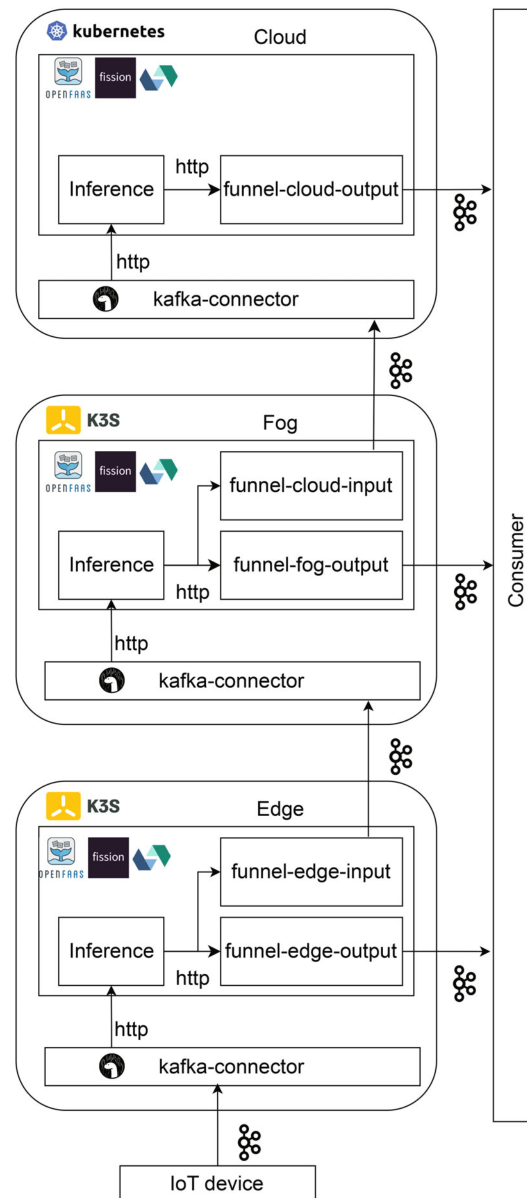

FIGURE 1 Architecture for DDNN inference with OpenFaaS, Fission, and not OpenWhisk.

through HTTP to each platform's gateway, triggering the desired function. However, during our research, we found out that OpenFaaS doesn't offer its Kafka connector with the free version OpenFaaS CE. As for Fission, we observed inconsistent behaviour in its Kafka connector, as it would disappear after re-deploying our application, and at times, required a complete platform reinstallation to function properly again.

To overcome this limitation, we created our own connector, namely `kafka-to-http`, which reads requests from Kafka and proxies them to the inference function. The connector is written in TypeScript and runs on the popular TypeScript/JavaScript runtime Deno.[18] Deno has been optimized for asynchronous tasks, particularly IO operations, which is most of the work our connector does. Thus, we expect our connector to perform well. Additionally, each layer has its own replica of the connector, and only proxies requests between its input topic and its inference function.

Finally, for our message format, we opted to use JSON for our system. While there are more compact or faster formats available, we find that JSON's human-readable nature and simplicity are beneficial for debugging and troubleshooting purposes. It allows us to quickly identify and resolve any issues that may arise, while still being fast enough for our use case.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3318 by Chua-Consorcio De Bibliotecas, Wiley Online Library on [14/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

4 | OPENFAAS

OpenFaaS is a container-based serverless platform featured on the Cloud Native Landscape, licensed under the MIT license. OpenFaaS also offers a paid version, OpenFaaS Pro, which builds upon the open-source project to deliver some additional features and commercial support.

There are multiple differences between both OpenFaaS Community Edition (CE) and OpenFaaS Pro. The most important one is the function autoscaler. The CE version uses what is called “Legacy scaling”, which imposes a maximum limit of 5 replicas[†] and it does not support scaling to zero. In addition, OpenFaaS claims that the autoscaler is for development only or internal use in non-business use cases.

One of the main features of OpenFaaS is the *templates*, which allows developers to set up a new OpenFaaS project in a given programming language. There are multiple programming languages officially supported, including popular ones such as Python, JavaScript, Ruby, and Java. These templates usually come with multiple *flavors* that include additional libraries or switch the base Docker image. There is also a vibrant ecosystem of community-maintained templates, which enables developers to create functions on other languages not officially supported by the OpenFaaS team.

OpenFaaS functions can be invoked using multiple triggers, such as HTTP, MQTT and S3 events. The platform also supports asynchronous invocations, in which events are stored on a NATS FIFO queue. One thing to keep in mind is that async requests execute sequentially, even if the requests are meant for different functions.

4.1 | Installation and developing an OpenFaaS function

OpenFaaS offers the flexibility to deploy its platform on various container orchestrators such as Kubernetes, K3s, OpenShift, or even on a single host using `faasd`. Nonetheless, we opted to deploy the full OpenFaaS platform on Kubernetes and K3s using Helm, which proved to be a simple and painless process, even for our ARM-powered cluster. The Helm chart’s default values include Prometheus metrics and a NATS queue for asynchronous function invocation, which are sensible choices. In just a matter of minutes, we were able to deploy our first function in a fully operational environment.

Additionally, the OpenFaaS installation guide provides instructions for installing the `faas-cli`, a Command Line Interface (CLI) that is available on Linux, Windows, and macOS. While this CLI is not essential, it is highly recommended as it allows developers to scaffold projects using the abovementioned templates, streamlining the development process.

OpenFaaS documentation[19] includes multiple tutorials and learning resources that ease the learning curve. For example, the “Your first OpenFaaS Function with Python” tutorial[20] is straightforward and covers step-by-step everything required to deploy a Python function with dependencies. The development workflow usually involves these steps:

1. **Pulling a template from the store**: Using `faas-cli`, a project can be easily scaffolded leveraging official and community templates.
2. **Modifying the source code**: Once the project is set up using the desired template, developers modify the source code of the function in order to archive the desired functionality. Each template includes documentation on how the function works and what the expected inputs and outputs are.
3. **Updating the OpenFaaS stack file**: Each function comes with a YAML description file used by `faas-cli` to deploy the function. The developer can enable certain configuration options by modifying this file. Commonly used options include environment variables, secrets, and Docker build options.
4. **Deploy the function to OpenFaaS Gateway**: Use the `faas-cli up` command to deploy the function. This step builds a Docker image that is pushed to a Docker Registry and later used by OpenFaaS to instantiate functions.

4.2 | Implementing our DDNN application in OpenFaaS

During our research, we found that there are no official or community templates available for DNN frameworks such as TensorFlow or PyTorch. As such, we decided to create our own TensorFlow templates for both `amd64` and `aarch64` architectures, based on the official `python3-http` templates. We use these architectures because, as we will see in the evaluation section, we have different infrastructures. Unfortunately, we were not able to create a GPU variant of the template since OpenFaaS does not officially support Kubernetes devices.

[†]This feature changed during the writing of this manuscript.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3318 by Chua-Consorcio De Bibliotecas, Wiley Online Library on [14/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

The source code for our `inference` function comprises a single Python file and the `.h5` model (trained model format from TensorFlow). To create our DDNN, we cloned the same function multiple times for each layer, replacing the `.h5` file accordingly. Although we could have set up a separate storage solution such as an S3 bucket, doing so would have increased our cold start latencies. Instead, we opted to include the necessary files directly in the function code. For our `funnel` function, we kept things simple: the function is the same for all three layers, based on the official `python3-http` template, and is composed of a single source file.

We were able to deploy our application easily on our `amd64` clusters. However, we faced some challenges with the ARM-based cluster. The problem arose when we tried to cross-compile from our local development machines (an `amd64` based machine) following the official OpenFaaS docs. We were using a hosted registry on our organization that had a self-signed TLS certificate, which was not being recognized by the `faas-cli`. While Buildx, the build backend used by `faas-cli`, supports custom CA certificates, `faas publish` does not offer pass-through options to Buildx. After trying different approaches and talking to the maintainers, we ended up using a Raspberry Pi as our image builder and using the standard `faas up` command to deploy the application.

5 | FISSION

Another Kubernetes-native serverless platform which is also featured on the Cloud Native Landscape, licensed under the Apache 2.0 license.[21] It offers an interesting concept called *environments*, which are a set of running containers with a small dynamic loader ready to launch functions. This allows for functions to start immediately, reducing the latencies for cold starts.

Environments are the language-specific parts of Fission. They are made up of a container with an HTTP server, and usually a dynamic loader (called *fetcher*) that can load a function. Some environments also contain builder containers, which take care of compilation and gathering dependencies.

Environments are currently available with two strategies:

1. **PoolManager**: Fission keeps a running set of "warm" pods from the selected environment. When a request arrives, the fetcher downloads the function and injects it into the environment. This pod will be used for subsequent requests and will be cleaned after a certain idle period. This strategy is convenient for functions that are short living and require short cold start times.
2. **NewDeploy**: Creates a Kubernetes Deployment with a service and a HorizontalPodAutoscaler (HPA)[22] for function executions. This enables the autoscaling of function pods and load balancing of the requests between pods. When a function experiences a traffic spike, the service helps to distribute the requests to pods belonging to the function. The HPA scales the replicas of the deployment based on the conditions set by the user. If there are no requests for a certain time, the idle pods are cleaned up. This strategy though increases the cold time of a function allows functions to serve massive traffic.[23]

Fission boasts of one key feature that sets it apart from other serverless platforms: its environments are full-featured Kubernetes pods. The platform fully embraces the orchestrator, allowing Fission to make use of various resources available in Kubernetes such as config maps, secrets, volumes, and even GPUs. With these resources at its disposal, Fission's environments provide developers with a lot of flexibility and enable them to create complex serverless applications that can take advantage of the existing capabilities of Kubernetes.

5.1 | Installation and developing a Fission function

Fission can be installed using Helm charts or Kubernetes objects, although Helm allows more features to be enabled. This is important, as the base Fission installation only includes the core components required for developing and testing functions. The default installation results in a smaller control plane, but requires the operator to enable each feature manually.

The installation guide[24] also provides instructions on how to install `fission`, a CLI available on Linux, Windows, and macOS. The CLI is required, as is the only way to operate Fission.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3318 by Chua-Consorcio De Bibliotecas, Wiley Online Library on [14/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Fission's documentation[25] is complete and describes all the main features of the platform. The developer team has curated a great set of examples for all of the languages they support, which makes developing a new function easy. The development workflow usually involves the following steps:

1. **Choosing the right environment**: Each function requires an environment to run. Users can create as many environments as they need with different options, but each function can only run on one environment.
2. **Scaffolding the project**: Each environment comes with its own builder, which is used to compile and install dependencies. The developer is responsible for reading the documentation and creating a project structure that matches the builder's requirements.
3. **Modifying the source code**: Once the project is ready, developers modify the source code to archive the desired functionality. The expected function interfaces of each environment are based on well-established libraries and frameworks.
4. **Deploying the environment and function**: The `fission` CLI is used to package the source code and send it to the selected environment builder. The builder is in charge of building the archive file that will be fetched by the fetcher pod on function execution. Fission doesn't create triggers for functions by default, which means that the function will be only available using the `fission fn test` command.

To automate function deployment, Fission offers *specs* which are Kubernetes Custom Resources Definition (CRD) files that the CLI applies. One advantage of this approach is that environments can be extended using Kubernetes Pod specs. This provides functions with access to volumes, environment variables, and sidecar/init containers that would be otherwise exclusive to fully featured Kubernetes pods.

Fission's unique approach to serverless has a high learning curve: developers must learn the basics of environments before creating functions, as well as read the instructions of the selected environment on how to set up the project. On the other hand, Fission's specs provide developers with a familiar interface which can be a big advantage for developers who already know Kubernetes.

5.2 | Implementing our DDNN application in Fission

Like OpenFaaS, Fission did not have any official TensorFlow or PyTorch environments. We created our own TensorFlow environments for both `amd64` and `aarch64` architectures based on the official `python-3.10` environment. This environment is CPU only, although we could create one that leverages the GPUs from our clusters.

We successfully ported our funnel and inference functions from OpenFaaS to Fission. Both functions exhibit the same behaviour described earlier. The funnel function uses the official `python-3.10` environment, while the inference function uses our new TensorFlow environment.

Deploying functions on Fission was a seamless experience for us. The platform automates the whole life cycle of a function, from source to pod, reducing the amount of manual work required from developers. Dependencies were installed automatically by the environment's builders, and artifacts were stored successfully by Fission's own package manager.

6 | OPENWHISK

A serverless platform from the Apache foundation, licensed under the Apache 2.0 license.[26] The project is the serverless platform of choice for IBM, which uses OpenWhisk as the machinery behind Cloud Functions.

6.1 | Installation and developing an OpenWhisk function

OpenWhisk offers many different deployment options: the platform can be installed on Kubernetes, Docker (with Docker Compose), Ansible, and Vagrant. This is one key feature of OpenWhisk, as the platform maintains the same functionality despite the installation method. We choose to install OpenWhisk on Kubernetes using the official Helm option.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3318 by Chua-Consorcio De Bibliotecas, Wiley Online Library on [14/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

The installation was challenging. We found that the installation guide[27] was incomplete and outdated. Thankfully, some issues we faced were documented on a public troubleshooting document.[28] The installation guide also provides instructions on how to install `wsk`, which is a CLI tool available on Linux, Windows, and macOS used to manage functions. This CLI is not mandatory, as the service can be fully managed through REST API calls.

OpenWhisk documentation includes minimal examples of how to deploy basic functions on all of their supported languages, including popular options such as JavaScript, Go, Python, and Java. These functions are usually composed of one single source file that is sent as plain text to the control plane. More complex functions, such as those that require additional dependencies, are allegedly possible but follow language-specific rules. The development workflow usually involves the following steps:

1. **Scaffolding the project**: Because the CLI `wsk` lacks any template functionality, the developer is in charge of scaffolding the project based on the docs for each supported language.
2. **Modifying the source code**: After scaffolding the project, developers modify the source code according to their requirements. The function interface, including expected inputs and outputs, is defined in the documentation for each language.
3. **Deploying the function**: OpenWhisk lacks a builder CI infrastructure, and thus the developer is in charge of any compilation or dependency bundling that the function may require.

6.2 | Implementing our DDNN application in OpenWhisk

In our attempt to recreate our application on OpenWhisk, we first started with a simple Python function with basic dependencies, notably the popular `requests` library. Following the recommended blog post "Python Packages in OpenWhisk",[29] we faced challenges with managing dependencies. Unlike the other platforms we tested, OpenWhisk requires developers to manually manage virtual environments inside Docker containers. This workflow is clearly error-prone and it should be automatically handled by the `wsk` CLI, not by the developer itself. After succeeding with the installation of dependencies, we obtained cryptic error messages we couldn't solve. After more failed attempts at running our simple Python function, we gave up the development of our application on this platform.

7 | EVALUATING SERVERLESS PLATFORMS FOR DDNN INFERENCE OVER THE CLOUD-TO-THINGS CONTINUUM

To evaluate the serverless platforms and their capabilities on resource constrained devices, such as those found on edge deployments, we conducted a benchmark using a simulated IoT device that publishes data to Kafka at a rate of one message per second, with the speed increasing every 100 messages up to 400 messages. Our objective is to observe the behavior of these platforms and verify their autoscaling capabilities under load. The benchmark aims to recreate a scenario where data is generated and needs to be processed quickly, such as those seen in real-world IoT applications. Although the deployment of serverless functions can present different challenges in edge devices due to their limitations, as part of the evaluation in this section we study the feasibility of these technologies in such infrastructures.

7.1 | Experimental setup

Our DDNN model was trained with Kafka-ML using the MNIST train dataset. The model is distributed with three layers following the BranchyNet architecture:[30] Cloud, Fog, and Edge. The Cloud receives input with a shape of 16 and has one output layer with 10 nodes, utilizing softmax activation. The Fog layer receives input with a shape of 32, processes it through a dense layer with a rectified linear unit (ReLU) activation function, and outputs to the Cloud layer. It also has an output layer with 10 nodes using softmax activation. Finally, the Edge layer receives input images with a shape of $28 \times 28 \times 1$ and processes them through a dense layer with a ReLU activation function. The output is then passed to the Fog layer, and it also has an output layer with 10 nodes using softmax activation. The models are trained using the TensorFlow framework, and each level is defined as a separate model. The TensorFlow version used for training is 2.7.0, while the version used for inference is 2.11.0. Unlike most image classification models that use convolutional neural

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3318 by Chua-Consorcio De Bibliotecas, Wiley Online Library on [14/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

networks, our model is composed of dense layers. The reason for this decision was to randomize the output of the model, as it had been performing exceptionally well with the original configuration.

Our Kubernetes clusters have the following configuration:

1. **On premise cloud cluster**: Seven state-of-the-art nodes. Each machine has an Intel(R) Xeon(R) Gold 6230R CPU with two NVIDIA(R) Tesla(R) V100 GPUs as well as 384GB of RAM. These nodes run Kubernetes v1.21.6 and Docker 20.10.8 on top of Ubuntu 20.04.3 LTS. A Kubernetes master was deployed in one node, while the remaining six are Kubernetes workers. We run Kafka using the `bitnami/kafka` Helm chart version 21.3.1, with two brokers and persistence disabled. This Kafka is in charge of the input topic of this layer. The number of partitions for the input topic is 1.
2. **Fog cluster**: A single node composed of Intel(R) Core(TM) i9-10900K CPU and 64GB of RAM. This node runs K3s v1.25.7+k3s1 on top of Ubuntu 21.04, with Traefik disabled. We run Kafka using the `bitnami/kafka` Helm chart version 21.3.1, with one broker and persistence disabled. This Kafka is in charge of the input topic of this layer. The number of partitions for the input topic is 1.
3. **Edge cluster**: Six Raspberry Pi model 3B+, with 1GB of RAM and a 64GB Samsung EVO Plus SD card. These nodes run K3s v1.25.7+k3s1, with Traefik disabled, on top of Raspberry Pi OS lite (64-bit) version 2023-02-21. We run Kafka using the `bitnami/kafka` Helm chart version 21.3.1, with one broker and persistence disabled. We had to bump the Zookeeper image to `3.8.1-debian-11-r8`, as it's the first 3.8 release with support for ARM. This Kafka is in charge of the input topic of this layer, the output topic from the cloud and the early exits of Fog and Edge layers. The number of partitions for these topics is 1.

For our serverless platforms, we are using OpenFaaS 12.0.2 with 4 gateway replicas on our Edge and Cloud clusters, and only one gateway replica on our Fog. For Fission, we are using v1.18.0 with InfluxDB enabled, persistence disabled and router deploy as daemon set enabled.

On both platforms, we set our maximum number of replicas to 20 for our inference function, and a minimum of 1 replica. Our funnel functions were configured with a maximum of 10 replicas and a minimum of 1. For CPU and memory, we provided our inference function with a maximum of 1000m of CPU and 1024MB of memory on our Fog and Cloud clusters. On the Edge, we set up our inference function with 500m of CPU and 512MB of memory.

7.2 | Analysis of results

Figure 2 describes our obtained results. We recorded the amount of time elapsed since we sent a message to Kafka, effectively measuring the Round Trip Time (RTT) for our system. OpenFaaS' Edge layer recorded the maximum response time at around 926 s. On the other hand, the minimum response time recorded was around 3 s, which was also recorded by OpenFaaS' Edge layer. The average response time for all topics ranges from around 430 to 439 s, with Fission's Cloud

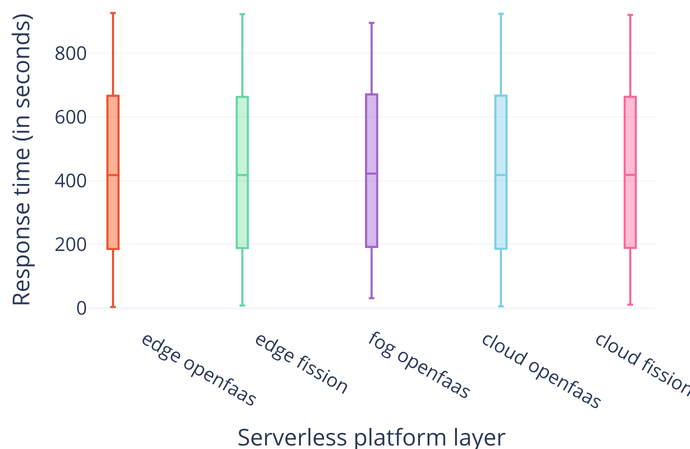

FIGURE 2 Box plots of requests latencies for a single client.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3318 by Chua-Consorcio De Bibliotecas, Wiley Online Library on [14/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

layer having the lowest average response time, and the OpenFaaS' Fog layer having the highest average response time. The median response time, which is less sensitive to outliers, ranges from around 417 to 422 s across all topics. The standard deviation, which measures the amount of variability in the data, ranges from around 270 to 274 s. High standard deviation values indicate that there is a significant amount of variability in the latency measurements.

Despite the high response times we were experiencing, OpenFaaS did not scale any of our functions on any of our layers. On the other hand, Fission was able to scale our inference function from one replica to two on our Edge layer. Even with the difference in the replica count, we did not measure significant differences in response times between both platforms. In fact, OpenFaaS had lower median and average response times compared to Fission.

7.3 | Discussion

After analyzing our logs, particularly OpenFaaS' Prometheus metrics, we discovered that our function scaling issue was caused by a low request per second on our gateways, which was about 0.4 requests per second (RPS). The slow data processing rate was unable to keep up with the rate of data input, leading to a general slowdown of our application. We concluded that this was primarily due to the limited resources available on our Raspberry Edge nodes, specifically the memory and disk speed. Our nodes had an average of only 100–150 megabytes of memory available. When the memory capacity was exhausted, the kernel resorted to swapping pages to the disk, which significantly slowed down our application.

Our Edge cluster is capable of running basic DDNNs models such as this one, but we concluded that the serverless platforms we were using may have overloaded our cluster. Specifically, both OpenFaaS and Fission deploy a considerable number of containers that act as the control plane of our functions. For example, OpenFaaS requires at least one gateway, an alert manager, Prometheus, and a queue worker with a NATS queue. Fission, on the other hand, requires a logger and router per cluster node, a builder manager, Kube watcher, and more. Additionally, each function in Fission includes a sidecar container (the aforementioned fetcher) which further increases the amount of resources required to run the platform. The findings highlight a key result of our study: none of the evaluated FaaS platforms were capable of running with restricted resources.

8 | REEVALUATING SERVERLESS PLATFORMS FOR DDNN INFERENCE: REMOVING THE EDGE

After analyzing our previous results, we opted for a different approach: merge the Edge and Fog layers in our system to evaluate the scalability of both platforms without the Edge layer, which represented a bottleneck. We also deployed our DDNN using Kafka-ML in order to compare the results between our original implementation and our serverless-based one. Our application and Kafka-ML differ in multiple places: where Kafka-ML uses a binary format for intra-layer communication, our application uses JSON. Additionally, Kafka-ML uses a monolithic architecture where the Kafka consumer, producer, and inference components reside in the same container, different from our application's architecture. Lastly, the TensorFlow versions are different: Kafka-ML uses version 2.7.0 while our OpenFaaS and Fission functions use version 2.11.0.

8.1 | Analysis of results

The results for a single client are depicted in Figure 3. It can be observed that OpenFaaS and Fission platforms maintained similar levels of response times. The median values for both platforms were around 230 ms for the Fog and 470 ms for the Cloud. OpenFaaS exhibited the lowest average, median, and minimum response times on both layers. Figure 4 presents the obtained latencies over time for both platforms on each layer. The latencies obtained by OpenFaaS remained mostly consistent across the benchmark, while Fission experienced bigger latency spikes. Notably, neither platform scaled the number of replicas during this benchmark. Compared to Kafka-ML's inference, OpenFaaS had the fastest inference time (213 s), followed by Kafka-ML (214 s), and lastly Fission (223 s).

However, as we increased the number of clients to three, we observed different behaviours from OpenFaaS and Fission. OpenFaaS scaled its inference function on the Fog up to five replicas, while Fission instantiated only two replicas. Surprisingly, Fission completed the test earlier than OpenFaaS, with less than half the number of replicas. Figure 5 shows

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3318 by Chua-Consorcio De Bibliotecas, Wiley Online Library on [14/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

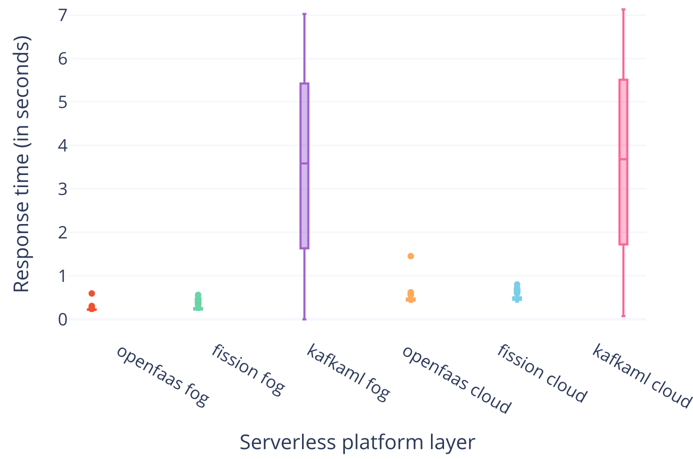

FIGURE 3 Box plots of request latencies for a single client.

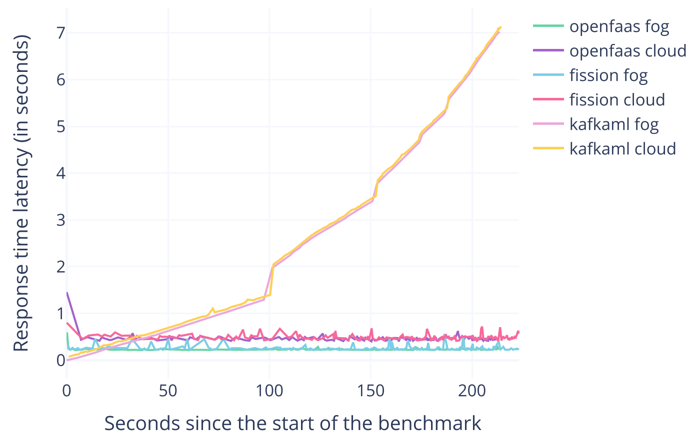

FIGURE 4 Latencies over time for a single client.

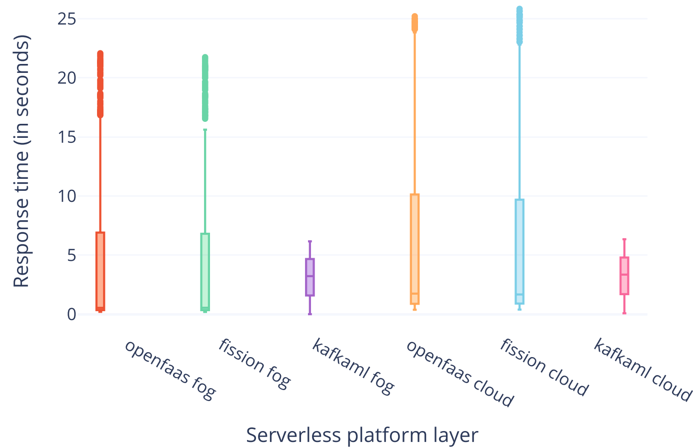

FIGURE 5 Box plots of request latencies for three clients.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3318 by Chua-Consorcio De Bibliotecas, Wiley Online Library on [14/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

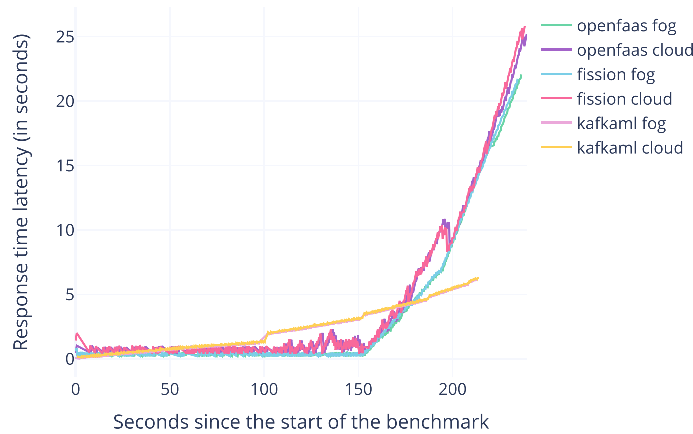

FIGURE 6 Latencies over time for three clients.

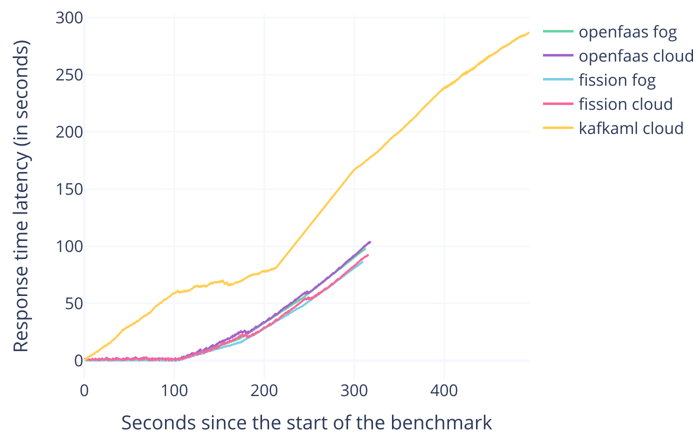

FIGURE 7 Latencies over time for five clients.

the results, indicating that both platforms had similar response times. Figure 6 presents the data over time, which also shows that our Kafka connector reached its limit in the third stage, at around 150 s. Remarkably, compared to Fission (238 s) and OpenFaaS (239 s), Kafka-ML had the shortest inference time (213 s).

Finally, we increased the number of clients to five, causing the connector to reach its limit shortly after the 100-second mark. OpenFaaS and Fission exhibited similar scaling behavior as before, with OpenFaaS having five replicas and Fission having two. Again, Fission completed the test faster than OpenFaaS. Figure 7 plots the data for this stage, while Figure 8 presents the results over time. We can notice how Kafka-ML did not produce outputs on the Fog layer due to how our DDNN model was defined (only Cloud early exit is activated). Fission finished the fastest at 315 s, followed by OpenFaaS at 318 s, and lastly Kafka-ML at 494 s. In this case, the serverless platforms clearly improve the response time of the monolithic architecture of Kafka-ML.

8.2 | Discussion

In the first round, with only one simulated device, we observed results that were consistent with Mohanty et al.[11] Specifically, we found that Fission's router component still had a significant number of outliers in terms of latency. The overhead added by the router was significant leaving the platform behind Kafka-ML and OpenFaaS. During this phase, we can also see how OpenFaaS shaved a second of Kafka-ML. This was due to small gains in parallelism: while our inference function was performing the computation, our Kafka connector was already pooling the brokers for the next value.

For the second round, with three simulated devices, we observed that both OpenFaaS and Fission were 10% slower than Kafka-ML. We believe this was likely due to the fact that both platforms created new instances of the inference

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3318 by Chua-Consorcio De Bibliotecas, Wiley Online Library on [14/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

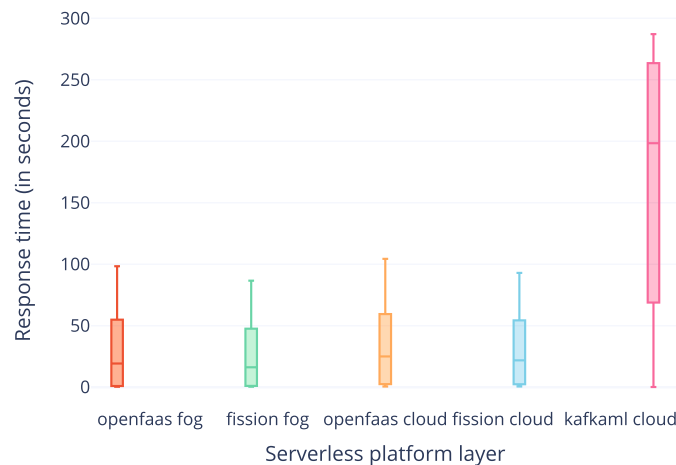

FIGURE 8 Box plots of request latencies for five clients.

function, leading to cold start latencies. Additionally, we suspect that a large amount of data transfers between our connector and the inference function may have contributed to the slower performance of the serverless platforms.

Lastly, increasing the number of clients to five showed that both OpenFaaS and Fission were able to perform significantly better than Kafka-ML. The results indicated a 55% in performance over Kafka-ML's DDNN. Fission was particularly noteworthy as it kept the number of replicas low and more stable, resulting in slightly faster performance compared to OpenFaaS.

Our experiments revealed that OpenFaaS exhibited a more aggressive scaling behaviour compared to Fission. OpenFaaS would frequently evict pods during the "ContainerCreating" phase, resulting in greater resource consumption, especially with big functions like ours. The "legacy scaler" of OpenFaaS was found to be the reason for this behaviour: the algorithm used by OpenFaaS CE depends heavily on Prometheus RPS metrics and alerts fired by Alert Manager. When a function starts to lag behind the number of requests, an alert is fired that increases the number of replicas by a given rate. However, when said alert stops firing, the gateway abruptly reduces the number of replicas to a minimum. Because our inference functions are long lived, it wasn't able to reach the expected RPS threshold at all times. As a result, our they were continuously scaled up and down.

On the other hand, Fission was more cautious in its scaling approach, only scaling our functions during high-traffic periods, and it would only evict pods after the function remained idle for some time. As a result, we consider Fission's autoscaler better suited for expensive and long-running functions rather than OpenFaaS.

About Kafka-ML, the framework lacks auto-scaling capabilities. This limitation resulted in a notable increase in latencies compared to OpenFaaS and Fission, primarily due to messages being kept in Kafka for extended periods, awaiting consumption by Kafka-ML's single inference pod.

Lastly, regarding the transfer costs between connectors, it is essential to clarify that such costs do exist in both OpenFaaS and Fission. However, in the five-client scenario, the increased workload and demand for processing have the effect of making these transfer costs less pronounced. This is due to the speed up obtained through multiple replicas, which effectively mitigates the impact of transfer costs in the context of higher client loads.

9 | CONCLUSION AND FUTURE WORK

In this paper we presented an overview of the three most popular open-source serverless functions, highlighting the installation process, potential benefits to the development experience and unique features of each platform. Additionally, we explored the potential of serverless and shared our experience of building a low-latency serverless DDNN IoT application.

We found that OpenFaaS and Fission are the most complete platforms, as they provide crucial abstractions which make the developer experience a breeze. For OpenWhisk, we conclude that the platform is unstable and unmaintained. Particularly, OpenFaaS had the best documentation and examples of all. The platform offers templates for scaffolding templates, a large collection of supported languages and a great developer experience thanks to its CLI. Deployment across the Cloud-to-Things continuum is simplified thanks to its stack files, although cross-compilation on self-hosted

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3318 by Chua-Consorcio De Bibliotecas, Wiley Online Library on [14/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

infrastructure can be cumbersome. Nonetheless, we found its business model a concern. OpenFaaS restricts access to critical features, such as the enhanced scheduler mentioned earlier, by keeping them closed-source. This may result in the unwanted vendor lock-in commonly observed with other proprietary solutions.

Fission is the most flexible platform, as it completely embraces Kubernetes features such as Kubernetes devices and volumes. This leaves the platform locked into Kubernetes, but given how ubiquitous Kubernetes has become we can hardly see it as a disadvantage. Fission also had a great developer experience, especially when building our function's source code. However, Fission was the heftiest platform, as it created more control pods and containers than OpenFaaS. Overall, we found that serverless requires more resources available than traditional computing. Therefore, we don't consider either OpenFaaS CE or Fission suitable options for constrained environments (i.e., Edge). Nonetheless, for non-constrained environments such as Fog and Cloud, serverless could be seen as an alternative to traditional monolithic deployments.

As for future work, we plan to explore the possibilities of serverless on other IoT fields and adapt the Kafka-ML framework to work natively with serverless computing. One potential line of research would be serverless stream processing across the Cloud-to-Things continuum, where these serverless platforms are used to process the incoming data efficiently with a similar architecture to what we proposed in this paper. Although previous works have been made on this area,[31] these platforms are limited to the AWS Lambda infrastructure.

AUTHOR CONTRIBUTIONS

Altair Bueno: Software development, Conceptualization, First manuscript draft. Bartolomé Rubio: Supervised the research, Conceptualization, Manuscript review, Funding. Cristian Martín: Helped with the deployment of Kafka-ML, Supervised the research, Conceptualization, Manuscript review, Funding. Manuel Díaz: Supervised the research, Conceptualization, Manuscript review, Funding.

ACKNOWLEDGMENTS

This work is funded by the Spanish projects TSI-063000-2021-116 ('Digital vertical twins for B5G/6G networks'), TED2021-130167B-C33 ('GEDIER: Application of Digital Twins to more sustainable irrigated farms'), PID2022-141705OB-C21 ('A framework for agnostic compositional and cognitive digital twin services (DiTaS)'), and CPP2021-009032 ('ZeroVision: Enabling Zero impact wastewater treatment through Computer Vision and Federated AI'). Funding for open access charge: Universidad de Málaga / CBUA.

CONFLICT OF INTEREST STATEMENT

The authors declare no potential conflict of interests.

DATA AVAILABILITY STATEMENT

The software and data that support the findings of this study are openly available in `faas-ddnn-continuum` at https://github.com/ertis-research/faas-ddnn-continuum.

ORCID

Altair Bueno https://orcid.org/0009-0001-3124-1742

Bartolomé Rubio https://orcid.org/0000-0002-8279-4224

Cristian Martín https://orcid.org/0000-0003-0988-591X

Manuel Díaz https://orcid.org/0000-0002-0625-2730

REFERENCES

1. Adzic G, Chatley R. Serverless computing: economic and architectural impact. Paper presented at: Esec/Fse 2017. Association for Computing Machinery; New York, NY, USA. 2017 884–889.
2. Lynn T, Mooney J, Lee B, Endo P. *The Cloud-to-Thing Continuum Opportunities and Challenges in Cloud. Fog and Edge Computing*. Palgrave Macmillan Cham; 2020.
3. Chaves AJ, Martín C, Díaz M. The orchestration of machine learning frameworks with data streams and GPU acceleration in Kafka-ML: a deep-learning performance comparative. *Expert Syst*. 2024;41(2):e13287. doi:10.1111/exsy.13287
4. Carnero A, Martín C, Díaz M. Portable motorized telescope system for wind turbine blades damage detection. *Eng Reports*. e12618. doi:10.1002/eng2.12618

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3318 by Cbua-Consorcio De Bibliotecas, Wiley Online Library on [14/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

5. Carnero A, Martín C, Torres DR, Garrido D, Díaz M, Rubio B. Managing and deploying distributed and deep neural models through Kafka-ML in the cloud-to-things continuum. *IEEE Access*. 2021;9:125478-125495.
6. Torres DR, Martín C, Rubio B, Díaz M. An open source framework based on Kafka-ML for distributed DNN inference over the cloud-to-things continuum. *J Syst Archit*. 2021;118:102214.
7. Martín C, Langendoerfer P, Zarrin PS, Díaz M, Rubio B. Kafka-ML: connecting the data stream with ML/AI frameworks. *Future Gener Comput Syst*. 2022;126:15-33.
8. Baldini I, Castro P, Chang K, et al. Serverless computing: current trends and open problems. In: Chaudhary S, Somani G, Buyya R. eds. *Research Advances in Cloud Computing*. Springer; 2017:1-20.
9. Harauzek D. *Cloud Computing: Challenges of Cloud Computing from Business Users Perspective - Vendor Lock-in*. PhD thesis. Jönköping University; 2022.
10. Jarczyk O, Gruszka B, Jaroszewicz S, Bukowski L, Wierzbicki A. *GitHub Projects. Quality Analysis of Open-Source Software*. Springer International Publishing; 2014:80-94.
11. Mohanty SK, Premsankar G, Francesco dM. An evaluation of open source serverless computing frameworks. Paper presented at: 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). 2018 115-120.
12. Balla D, Maliosz M, Simon C. Open source FaaS performance aspects. Paper presented at: 2020 43rd International Conference on Telecommunications and Signal Processing (TSP). 2020 358-364.
13. Palade A, Kazmi A, Clarke S. An evaluation of open source Serverless computing frameworks support at the edge. Paper presented at: 2642-939X of 2019 IEEE World Congress on Services (SERVICES). 2019 206-211.
14. Jindal A, Gerndt M, Chadha M, Podolskiy V, Chen P. Function delivery network: extending serverless computing for heterogeneous platforms. *Softw Pract Exp*. 2021;51(9):1936-1963. doi:10.1002/spe.2966
15. Park S, Choi J, Lee K. All-you-can-inference: Serverless DNN model inference suite. Paper presented at: WoSC'22. Association for Computing Machinery; New York, NY, USA. 2022 1–6.
16. Carreira J, Fonseca P, Tumanov A, Zhang A, Katz R. Cirrus: a serverless framework for end-to-end ML workflows. Paper presented at: SoCC'19. Association for Computing Machinery; New York, NY, USA. 2019 13–24.
17. Yu M, Jiang Z, Ng HC, Wang W, Chen R, Li B. Gillis: serving large neural networks in serverless functions with automatic model partitioning. Paper presented at: 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS), July 7-10, 2021, Virtual Washington DC, USA. IEEE. 2021 138–148.
18. Deno's official website. https://deno.land/. 2023.
19. OpenFaaS docs. https://docs.openfaas.com/. 2023.
20. OpenFaaS docs. Your first OpenFaaS Function with Python. 2023 https://docs.openfaas.com/tutorials/first-python-function/
21. Github. Fission/fission: Fast and simple serverless functions for Kubernetes. 2023 https://github.com/fission/fission
22. Kubernetes docs. Horizontal Pod Autoscaling. 2023 https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
23. Fission docs. Executor. 2023 https://fission.io/docs/architecture/executor/
24. Fission docs. Installing Fission. 2023 https://fission.io/docs/installation/
25. Fission docs. https://fission.io/docs/. 2023.
26. Apache/openwhisk. Apache OpenWhisk is an open source serverless cloud platform. 2023 https://github.com/apache/openwhisk
27. apache/openwhisk-deploy-kube/docs/k8s-diy.md. https://github.com/apache/openwhisk-deploy-kube/blob/e202f469181716ba15e0676245680057d294d951/README.md. 2023.
28. openwhisk-deploy-kube/docs/troubleshooting. https://github.com/apache/openwhisk-deploy-kube/blob/e202f469181716ba15e0676245680057d294d951/docs/troubleshooting.md. 2023.
29. Python Packages in OpenWhisk. https://jamesthom.as/2017/04/python-packages-in-openwhisk/. 2023.
30. Teerapittayanon S, McDanel B, Kung H. BranchyNet: fast inference via early exiting from deep neural networks. Paper presented at: 2016 23rd International Conference on Pattern Recognition (ICPR). 2016:2464-2469.
31. Werner S, Girke R, Kuhlenkamp J. An evaluation of Serverless data processing frameworks. Paper presented at: WoSC'20. Association for Computing Machinery; New York, NY, USA. 2021:19-24.

How to cite this article: Bueno A, Rubio B, Martín C, Díaz M. Functions as a service for distributed deep neural network inference over the cloud-to-things continuum. *Softw: Pract Exper*. 2024;1-15. doi: 10.1002/spe.3318

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3318 by Chua-Consorcio De Bibliotecas, Wiley Online Library on [14/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License