A Cloud enabled CNN as a Function-as-a-Service

1st Vincenzo Bevilacqua
University of Naples Parthenope
Naples, Italy
vincenzo.bevilacqua001@studenti.uniparthenope.it

2st Antonio Di Marino
University of Naples Parthenope
Naples, Italy
antonio.dimarino001@studenti.uniparthenope.it

3st Michele Zito
University of Naples Parthenope
Naples, Italy
michele.zito001@studenti.uniparthenope.it

***Abstract*—Recent advances have led to the consolidation of function-as-a-service (Faas) in cloud services. In this paper we will demonstrate how it has been possible to use this technology to offer a fruit ripening state classifier as a service in the cloud, using Docker, Kubernetes and OpenFaas developing also a small web application for testing the responses by the running function itself.**

***Index Terms*—OpenFaas, Faas, Prometheus, Grafana, Cloud Computing, Kubernetes, Docker, Deep Learning, Cnn, YOLO, AlexNet, Flask**

I. Introduction

Function-as-a-Service is a technology that enables the serverless use of a function, giving developers the opportunity not to write monolithic code, but to implement modular functions. This serverless functionality allows to scale quickly by allocating only the resources needed on the server as opposed to using a virtual machine, greatly reducing cloud service costs. There are several providers that give developers the tools to write functions as a service, that are: Amazon Web Services Lambda, IBM Cloud Functions, Google Cloud Functions and Azure Functions.
The choice for this work is OpenFaas [1], an OpenSource framework for implementing Function-as-a-Service (FaaS). The reason behind the choice for an OpenSource framework is the possibility to have more flexibility in writing the function, without the constraints imposed by providers.
Although in other works the focus is on the performances of OpenFaas, in this one we wanted to use a Deep Learning model to be implemented as a faas, thus allowing serverless predictions to be made.
In the following sections we will discuss the implementation of the function itself and the web app made for testing, then a brief evaluation on the results, then we will talk about some related works that have been made and in the last section we will talk about some concluding remarks and future directions.

II. Making a CNN serverless

As a starting point for this work, we considered a previous work from some of the authors where they developed a Convolutional Neural Network based on YOLO [2] and AlexNet [3] for the classification of the ripeness of bananas. The identified classes for this model are seven: underripe, barely ripe, ripe, very ripe, overripe, rotten. [see Fig. 1]

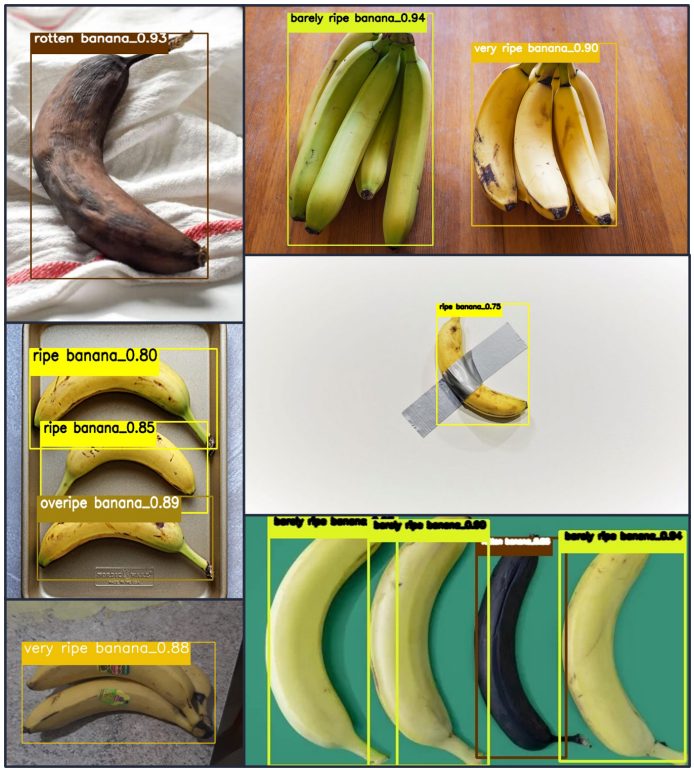

Fig. 1. Example of labels produced by the model.

The model uses YOLO for the multi-detection of bananas in images, and uses the AlexNet model for the classification of each image crop produced by the previous step [see Fig. 2].

The use of this model in this work has to be taken just as an example of the use of Deep Learning models used as FaaS.

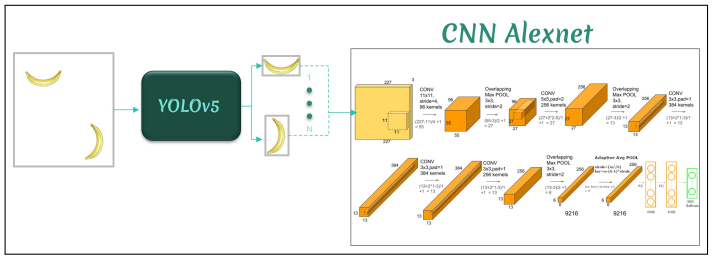

Fig. 2. Model architecture

The focus of this work is to evaluate the performance of this pre-trained model in a cloud environment, specifically as a serverless function, as mentioned above.

A. The Function-as-a-Service

Among the different providers for deplyoing functions-as-a-service, for this work the choice fell on OpenFaas. It makes it easy for developers to deploy event-driven functions and microservices to Kubernetes placing code or an existing binary in a Docker image to get a highly-scalable endpoint with auto-scaling and metrics. The user-defined functions can be accessed through a Gateway module implemented by OpenFaas. OpenFaaS defines an Alert-Manager that manages the scaling of the function instances, it monitors the Prometheus data-store that collects runtime metrics, and, according to these metrics, executes scaling events. In the Premium version of OpenFaaS there's also support for scaling the number of unused functions to zero, to save computing resources; in this work we used the base version of OpenFaas that requires one instance as minimum. The function instances are implemented by using a tiny Golang-based webserver, the Watchdog, that manages the forwarding of requests to the user-defined functions.

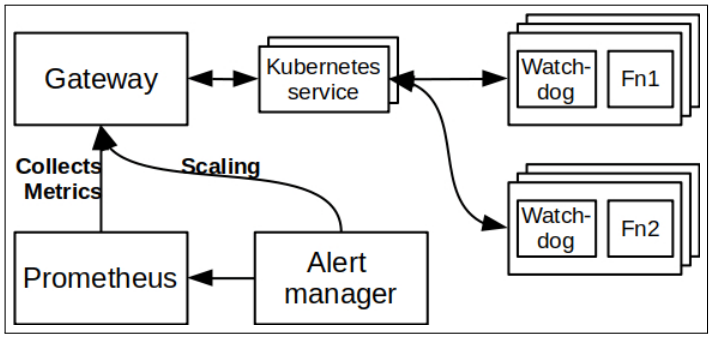

Fig. 3. OpenFaas architecture [4]

Once the environment is set up, a function handler and a configuration file in the .yml format must be written before the building, the pull on Docker Hub and the deploying on the cloud machine using Kubernetes.
The function handler is a Python file that can be summarized as follows:

Function 1 Handler

```
procedure HANDLE(request)
    if BAD request then
        return JSON_dumps(Error)
    end if
    img ← request['img']
    img_opencv ← OpenCV.decode(img)
    result ← Model.detect(img_opencv)
    encoded ← base64.encode(result)
    return JSONified(encoded)
end procedure
```

In the configuration file we set the gateway address for the function, its name, pydatascience-web [5] as its template, the folder where to find our function code, the Docker image name for building and some environment and autoscaling variables. After the building and the push on Docker Hub, the docker image was then placed on the GARR cloud machine, ready for execution.

B. The Web Application

To test this serverless function, HTTP requests must be sent to it, to do so we wrote a simple Flask based web app to load an image to classify and display the FaaS response. This web application is used just as an example of use, because requests could be sent in various ways, as long as they are encoded in the right way.
Once the user has loaded the image he wants to classify, an ajax query is called that requests to the running flask application to contact the serverless function; here the image loaded is encoded in a dictionary that is used in the HTTP request to the FaaS. Once the response is obtained, the image stored in the 'img' field is then returned to the ajax query that then displays it on the web page.

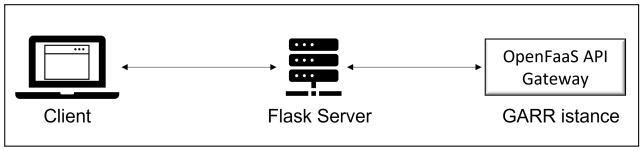

Fig. 4. Execution Order of Requests

III. Evaluation

We evaluated the performances of this OpenFaas function on a cloud machine provided by an italian research network, known as GARR [7], using Prometheus and Grafana to monitor some metrics on the requests.

A. Experimental Setup

The machine that we have instanced uses Ubuntu 20.04 as its operating system, with a RAM of 8 GB and 4 VCPU and 80GB of disk memory; this cloud machine is running on the Palermo branch of the research network and we have defined some network rules to make this instance available on a public IP.

As a first step we installed Kubernetes by remote connection, using k3sup [10], a tool that allows to install and access a kubernetes cluster via an ssh connection, using the private key created on the setup of the machine itself. With the Kubernetes cluster running, we then installed OpenFaas. On a local machine we defined the configuration file and the handler function [see Function 1] with all the dependencies, with all the model files required, as before and then we proceeded with the OpenFaas build, which installs all the required libraries. After the successful build, we pushed this image of the function on Docker Hub and then we deployed it on the Kubernetes cluster. As a final setup step we created and set an OpenFaas url to make it openly accessible.

B. Monitoring and Testing

To make possible to monitor the performances of the functions, we used Prometheus' metrics and Grafana to see them at runtime.
Prometheus, a Cloud Native Computing Foundation project, is a system for monitoring systems and services. It collects metrics from configured targets at given intervals, evaluates rule expressions, displays the results, and can trigger alerts when specified conditions are observed [6].
Grafana is an open-source platform for monitoring and observability, and allows to query, visualize, alert on and understand metrics, it also allows to create dynamic and reusable dashboards [8].
These both run on the Kubernetes cluster, to access them we made some port-forwarding on a local machine and then open the Grafana dashboard on a browser.
The metrics we wanted to monitor were: the number of total invocations since the function went running; the scaling of function replicas, to manage the number of requests; the execution duration, in seconds and in milliseconds, on the cloud machine; the function rate of the number of requests per second.

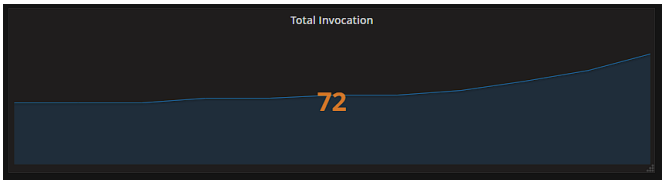

Fig. 5. Example of Number of total invocations

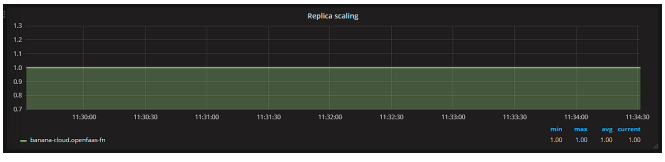

Fig. 6. Example of scaling

Fig. 7. Example of execution duration (s)

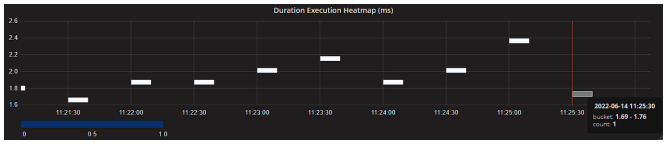

Fig. 8. Example of execution duration (ms)

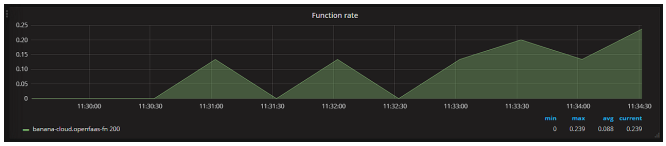

Fig. 9. Example of function rate

C. Autoscaling

OpenFaas can be used in two different ways, with a PRO edition or with a Community Edition. As we used the Community Edition, the autoscaling provided is just about the number of function invocations while in the PRO edition it can be possible to set up a scaling based on CPU and RAM parameters defined by the users.
Instead of using the CE autoscaler of OpenFaas, we used Metric Server [11], a Kubernetes package that collects resource metrics from Kubelets and exposes them in Kubernetes apiserver and lets use them by the Horizontal Pod Autoscaler [12].
To use the Metric Server, we edited the configuration file of the function, disabling the OpenFaas autoscaler and defining the minimum CPU value required to use the autoscaling.
To use the Metric Server we created another configuration file, where we defined the rules and the thresholds to enable the scaling, that will then be used by Kubernetes.
We decided to define rules based on the percentage of usage of the CPU, but there can be also defined other rules based on RAM usage.
To test the autoscaling of the function with these rules, we then simulated the case when it is receiving thousands of requests.

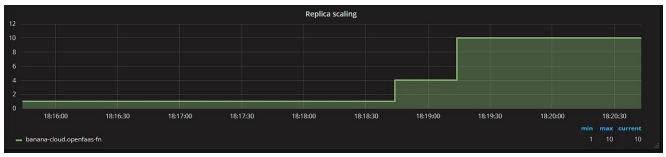

Fig. 10. Example of autoscaling

Fig. 11. Example of autoscaling

IV. Related Work

As a starting point for this work, we got inspiration from the work of S. K. Mohanty, G. Premsankar, M. Di Francesco [9] where they evaluate open source serverless computing frameworks like OpenFaas, Kubeless or Fission, carrying comprehensive feature comparison of these most popular frameworks, then they found that OpenFaaS has the most flexible architecture with support for multiple container orchestrators and finally evaluated the performances when deployed on a Kubernetes cluser characterizing the response time and success ratio for the deployed functions.
We also got inspiration from the work of D. Balla, M. Maliosz, C. Simon [4] where they showed the performance differences between the language runtimes of the FaaS platforms (Fission, Kubeless, OpenFaas), finding that the performance depends on the type of workload (compute or IO intensive). They also examined the supported auto-scaling algorithms and how the performance of each function runtime changes with auto-scaling.

V. Conclusion

In this work we took a pre-trained Convolutional Neural Network model that classifies the ripeness state of bananas and we made it serverless as a Function-as-a-Service using OpenFaas. In doing so we gained some advantages: the images to classify aren't saved locally on disk, but only kept in memory as we have defined the function handler that only takes images in a HTTP request. We could say that the model is encapsulated, i.e. we make a Deep Learning model available as simple as contacting an url via HTTP requests.
The set up and the installation of all of this is also flexible, because developers don't have to manually transfer code to online clusters, but they only have to manage remotely the deploying of the function itself using OpenFaas and k3sup on their local machines, while the clusters are provided by third party companies.

Acknowledgment

The deployment and the testing of this work was made possible thanks to the GARR, an italian research network, for making cloud resources available to the students of the University of Naples Parthenope.
The GitHub page for this work is at https://github.com/BananaCloud-CC2022-Parthenope/BananaCloud.

References

[1] "OpenFaas", https://github.com/openfaas/faas, (Accessed: 06/12/2022)
[2] "YOLOv5", https://github.com/ultralytics/yolov5, (Accessed: 06/12/2022)
[3] A. Krizhevsky, I. Sutskever, G. E. Hinton , "ImageNet Classification with Deep Convolutional Neural Networks", Curran Associates, Inc., 2012
[4] D. Balla, M. Maliosz, C. Simon, "Open Source FaaS Performance Aspects", 2020
[5] "pydatascience-web", https://github.com/LucasRoesler/pydatascience-template , (Accessed: 06/12/2022)
[6] "Prometheus", https://github.com/prometheus/prometheus , (Accessed: 06/12/2022)
[7] "Cloud GARR", https://cloud.garr.it/ , (Accessed: 06/12/2022)
[8] "Grafana", https://github.com/grafana/grafana , (Accessed: 06/12/2022)
[9] S. K. Mohanty, G. Premsankar, M. Di Francesco, "An evaluation of open source serverless computing frameworks", 2018
[10] "k3sup", https://github.com/alexellis/k3sup, (Accessed: 06/12/2022)
[11] "Metric Server", https://github.com/kubernetes-sigs/metrics-server, (Accessed: 06/12/2022)
[12] "Horizontal Pod Autoscaling", https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, (Accessed: 06/12/2022)