**JST: Smart Systems and Devices**
*Volume 35, Issue 1, January 2025, 001-009*

# Evaluation of the Scalability and Performance of the Open Source Serverless Computing Platforms

***Nguyen Tai Hung***[*]
*School of Electrical and Electronic Engineering, Hanoi University of Science and Technology, Ha Noi, Vietnam*
[*]*Correspoding author email: hung.nguyentai@hust.edu.vn*

**Abstract**

*The most powerful cloud computing service model currently being developed is the Serverless computing model. This model brings scalability and cost optimization in the process of deploying applications on cloud infrastructure. The application will be divided into functions that run a specific logic and those functions will be deployed as independent units on the Serverless computing platform. One of the strongest points of deploying applications running on the Serverless computing platform is its scalability and good processing performance. Scalability is demonstrated through the process of recognizing the actual usage needs of the function, from which the controllers in the Serverless computing platform will calculate and coordinate resources in the cloud environment appropriately, ensuring both service availability and saving idle resources. The performance of functions deployed on the Serverless computing platform mainly comes from the computing and storage capacity provided by the infrastructure, but the Serverless platforms also participate in supporting the process of optimizing the operating flow to minimize processing time and return response results to requests from the function caller. In this paper, we will build an experimental model with the two most popular open source Serverless computing platforms in the cloud computing development community, OpenFaaS and Knative. The purpose of this work is to compare and evaluate the scalability and performance in the process of operating applications on the open source Serverless computing platform. These two platforms rely on two different parameters to decide on the number of function instances.*

Keywords: Serverless, cloud, OpenFaaS, Knative.

## 1. Introduction

Cloud computing service providers divide into many different service provision models to increase the choice for their customers. Therefore, software developers will base on the characteristics of the architecture they are building to decide which service model to use to be most suitable and save costs and resources.

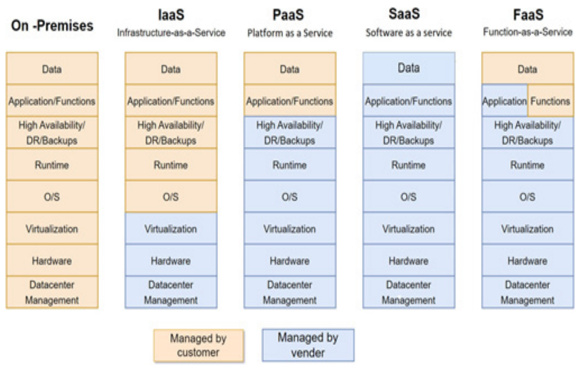

Fig. 1. Comparison of cloud computing service delivery models

Fig. 1 [1] illustrates the comparison between different cloud models which give the division of responsibility and roles between the vendor and the customers at various levels. While On-premise, Infastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) are traditional cloud based software deployment models, Function as a Service (FaaS) is the new one that proposed recently.

Serverless cloud computing, also known as "FaaS", is a new and exciting emerging model for deploying cloud applications, largely due to the recent shift in application architecture that enterprises are deploying to run on containers and build on the micro-services model [2]. Serverless is a cloud-native development model that allows developers to build, run, and deploy applications without the effort of managing and monitoring servers. Serverless still uses servers to run the developer's application, but the underlying infrastructure to provide resources for the server to run is abstracted or hidden from the developer's application development process.

From Fig. 2, it can be observed that when the system scale is expanded to meet the increasing demand from users, the Serverless model has optimized the use of resources very well and it is significantly more economical than building and deploying a separate server. The resources that are really needed during the running of the application or service of the tenant are allocated and the cost will only

ISSN: 2734-9373
https://doi.org/10.51316/jst.179.ssad.2025.35.1.1
Received: Aug 2, 2024; revised: Aug 22, 2024;
accepted: Sep 4, 2024.

be calculated on that amount of resources. In addition, when the demand decreases, the idle resources that are not really involved in the running of the application will be recovered to cut costs.

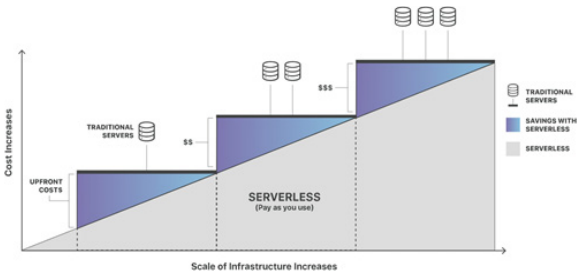

Fig. 2. The cost benefits of Serverless model over traditional server deployments

In more details, FaaS is a cloud computing service delivery model that allows users to execute their code in response to event calls. This service frees software developers from the complex cloud infrastructure management often associated with building and launching microservices applications [3]. Publishing a software application or web application on the internet often requires the provisioning and management of a physical or virtual server and the management of an operating system and web server. With the advent of FaaS, the key components of the cloud infrastructure such as physical hardware, virtual machines, operating systems, and web servers are automatically managed by the cloud service provider. Thanks to this, software developers only need to focus entirely on the individual functions of their application source code. In the FaaS model, functions are deployed and run on the cloud platform and are triggered by events or requests from users. FaaS focuses on running individual functions in isolation and independently, each function performing a specific job. When an event or request is triggered, the FaaS system automatically deploys an instance of the corresponding function and executes it. When the job is completed, the function instance is deleted and the resources are released.

Serverless model supposedly inherit some superior characteristics compared to other traditional cloud computing models, typically:

1. Elasticity: With FaaS, developers can focus on writing logic code for each individual function instead of building and managing the entire application. They can take advantage of these functions to handle events or requests from different sources automatically and flexibly.

2. Automatic scaling: The FaaS model automatically expands and shrinks the number of function instances based on actual requests. When an event arrives, the system will automatically deploy and run the necessary function instances to handle the request, and then automatically reduce the number of instances when no longer needed.

3. Scalability: FaaS provides automatic horizontal scaling for functions. Functions can be deployed across multiple server nodes to meet high demand, while taking advantage of the automatic scalability of the FaaS platform.

However, there're little analysis details and evaluation of Key Performance Indicators (KPIs) of the Serverless platform in the condition of real deployment given the fact that there're currently plenty of frameworks and platforms, both commercial and open sources, in the market and all claims of their superior. That's why on this work we have setup a Test-bed in our lab to test and compare the two main factors of the FaaS platform, namely the Scalability and Performance. We first will define the KPI parameters set and then setup an experimental environment to measure them in the high load conditions.

The contributions of this work are as the follows:

1) Proposition of a benchmarking frame work and methodology to measures scalability and performance KPIs of the (open sources) Serverless platforms;

2) First ever work done on comparison of the Serverless platforms on two very crucial perspectives of Scalability and Performance

The rest of this paper is organized as follows. Section II addresses related works. Section III presents our methodology and setup for benchmarking the scalability and performance KPIs between the two most popular Serverless platforms: Knative and OpenFaaS. Section IV discusses the experimental results and evaluations. Finally, Section V concludes the work.

## 2. Related Works

To deploy any cloud computing service model, it is necessary to have a data processing centre (Data-Center) including hardware devices that provides huge computing and storage capacity such as CPU, RAM, Storage, Switch, Router, ... Data centers always ensure timely and continuous provision of necessary resources for the platforms running above. In addition, cloud computing service providers also build distributed data centers located in different regions,unning independently and separately from each other to create redundancy and multi-region.

Choosing to use Serverless services between open source platforms or between service providers depends on many factors such as: project nature, popular requirements, platform popularity and support from the user community. Currently, the choice of Serverless service technology from service providers is much more popular than open source Serverless services because it provides more advanced features as well as a richer accompanying cloud computing service ecosystem. Serverless computing services of cloud computing service providers such as AWS, Google Cloud Platform, Microsoft Azure can be integrated with many other auxiliary services to link into a complete system such as event notification service, distributed database storage and management service, big data analysis, etc. However, the closed architecture of commercial Serverless computing platforms brings many limitations to software developers. To study and better understand how a Serverless computing platform works, our work will focus on popular open source Serverless computing architectures in the development community.

Among the popular open source Serverless computing platforms in the community [3], the three most advanced and still updated platforms are Knative [4], OpenFaaS [5], and OpenWhisk [6]. All three platforms run Serverless functions in a separate Docker container to isolate processes from each other. In addition, Knative and OpenFaaS require a container orchestrator to manage the networking and lifecycle of containers, while OpenWhisk does not need to be deployed on a container orchestrator. Knative is an open source project managed by the Cloud Native Computing Foundation (CNCF). It provides a platform for building and deploying Serverless applications on Kubernetes. Knative combines technologies such as Kubernetes, Istio, and Tekton to provide Serverless application management and scalability. It supports autoscaling, continuous deployment, and event management. OpenWhisk is an open source project of the Apache Software Foundation (ASF). It is a Serverless computing platform built on Kubernetes and is multi-language capable. The expansion and contraction of functions are managed directly by the OpenWhisk orchestrator. Finally, OpenFaaS with its powerful and flexible integration with Docker and Kubernetes, provides a command-line interface (CLI) that makes it easy for system administrators to develop and deploy functions on this platform. Software developers only need to provide the logic, source code, and dependencies of the function, while CLI handles the steps in the process of packaging the function into a Docker container and managing the lifecycle of this container through the Pod object in Kubernetes.

OpenFaaS is an open source Serverless platform (Fig. 3) built on Kubernetes. It allows application developers to easily deploy and manage functions flexibly without having to worry about the underlying infrastructure or maintain or operate the system. With OpenFaaS, application developers can write source code for small, single functions, and then OpenFaaS Operators are responsible for packaging them into Docker containers and deploying them to OpenFaaS. OpenFaaS will automatically manage the deployment and launch process of functions, automatically scaling up or down resources according to actual needs. OpenFaaS helps developers easily deploy event-driven functions and build applications in a microservice architecture for Kubernetes without having to configure and install complex systems. In addition to packaging source code or binary files, existing libraries in Docker, OpenFaaS also supports automatic creation of endpoints for each function so that users can easily send service requests to each function, and also makes it easy to monitor the status of the function, thereby makes decisions about expanding or shrinking copies of the function.

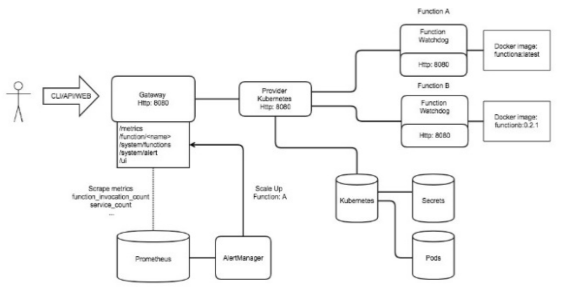

Fig. 3. Overall architecture of the OpenFaaS platform

Knative is an open source project of (CNCF, the largest organization in the field of cloud computing, developed on the Kubernetes platform. Knative provides a framework and a set of tools for building and operating Serverless computing applications on Kubernetes. The main goal of Knative is to help developers build and deploy Serverless applications easily and flexibly. It provides resource management, automatic scaling, application version management, and event handling capabilities. Knative helps create a reliable and scalable Serverless environment on the Kubernetes infrastructure. Knative's architecture consists of two main components: Knative Serving and Knative Eventing. Knative Serving is the component responsible for managing the deployment and running of service functions on the Kubernetes platform. It provides automatic scaling and load balancing based on workload and incoming requests. Knative Serving (Fig. 3) uses the concept of revision to manage application versions and helps route requests to specific versions [7]. It also supports concepts such as route and configuration to manage application routing and configuration. Knative is the best integrated Serverless computing platform with K8S today. Thanks to the high management and customization capabilities that the K8S ecosystem brings, it will be very meaningful for Knative to operate the application developer's functions in the most stable way. Knative has built controllers according to K8S standards along with defining many custom resources (Custom Resource Definition - CRD) that interact with K8S core resources to take advantage of the scalability, scaling of the number of Pods executing functions or the intelligent load balancing capabilities that K8S has available. The Knative Serving architecture also defines a set of custom resource objects (CRDs), which are used to identify and control the flow of function requests sent to each specific instance, or in other words, coordinate the entire process of handling request traffic to the system. The main CRDs of Knative Serving integrated into K8S include services, routes, configurations, and revisions.

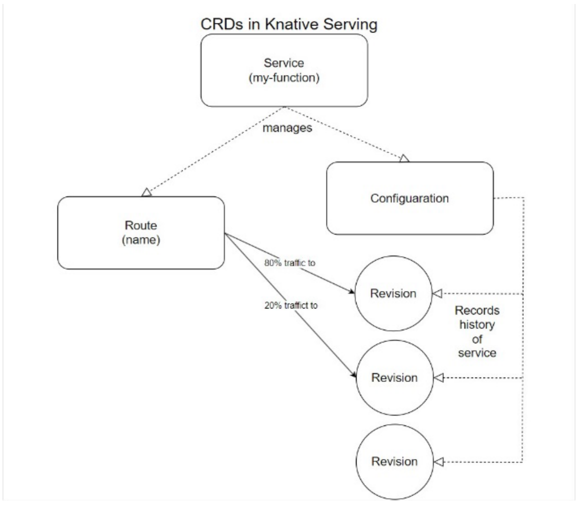

Fig. 4. Block diagram of components in Knative Serving

From the block diagram in Fig. 4, it can be seen that the Service object has a management and coordination role for other objects in Knative Serving. Service is the basic object for deploying Serverless functions in Knative. A Service represents a specific version of the application and defines how the application will be deployed and scaled [4]. The Service is connected to a container image or a builder tool to create an execution environment for the application. Route is the object that defines how to reach and route requests to specific versions of the Service. It provides a simple mechanism for managing application versions, distributing traffic according to the configuration to exactly that version. Configuration is a resource for configuring the properties of a Service [7]. It allows the user to define how the Service will behave, the parameters configured in Configuration include the function version number, environment configuration, network configuration, and many other properties. Revision is a representation of a specific version of a Service. Each time a Service is deployed or updated, a new Revision is created [7]. The Revision includes all the information and resources required to deploy the application. The Service's controller keeps track of the state of the Configuration and Route it owns, reflecting their state and conditions as those of the Service itself. The Configuration's Ready conditions are exposed through the Service's "ConfigurationsReady" condition. The Route's Ready conditions are exposed through the Service's "RoutesReady" condition.

OpenWhisk is another open source integration platform for building Serverless computing architectures. It allows software or web developers to run code without having to worry about infrastructure management, automatically scale, and pay based on actual resource usage. OpenWhisk is developed by the Apache Software Foundation and supports multiple programming languages, including Node.js, Python, Java, Swift, and Go. It helps developers build and deploy applications and services flexibly and efficiently. More about the OpenWhisk can be found on [3, 6].

Peer work on the OpenLambda platform presents an analysis of the scaling advantages of serverless computing, as well as a performance analysis of various container transitions [8]. Other performance analyses have studied the effect of language runtime and VPC (Virtual Private Cloud) impact on AWS (Amazon Web Services) Lambda start times [9], and measured the potential of AWS Lambda for embarrassingly parallel high performance scientific computing [10]. Serverless computing has proved a good fit for IoT applications, intersecting with the edge/fog computing infrastructure conversation. There are ongoing efforts to integrate serverless computing into a "hierarchy of datacenters" to empower the foreseen proliferation of IoT devices [11]. AWS has recently joined this field with their Lambda@Edge [12] product, which allows application developers to place limited Lambda functions in edge nodes. AWS has been pursuing other expansions of serverless computing as well, including Greengrass [13], which provides a single programming model across IoT and Lambda functions. Serverless computing allows application developers to decompose large applications into small functions, allowing application components to scale individually, but this presents a new problem in the coherent management of a large array of functions. AWS recently introduced Step Functions [16], which allows for easier organization and visualization of function interaction.

Our investigation results also show that even though with many Serverless platforms as mentioned, there are none of project that focus on evaluation of their performance and scalability, until recently. That motivated us on proceeding with this work to deeply investigate their performance KPIs in order to prove that Serverless (or FaaS) is more scalable than those of the traditional cloud computing models.

## 3. Proposition of Methodology to Investigate the Scalability and Performance of Open Source Serverless Platforms

On this section, we will focus on presenting the design and implementation process of methodology to investigate the scalability and performance of open source Serverless computing architectures. The deployment of Serverless computing platforms will be carried out on the Kubernetes container orchestration platform built from the OpenStack virtualization. We will then measure system parameters to evaluate the scalability or resource recovery of applications running on the Serverless computing platform as well as the performance during request processing. The measurement parameters will be focused on the processor consumption for purpose of easy monitoring.

### *3.1. Proposed Evaluation Methodology*

In the actual operation of service functions, the processing needs from external sources are unpredictable, monitoring the load and manually configuring the resizing of the functions are very laborious and not feasible. Therefore, we have to make our effort to build controllers (based on the open source projects) with coordination objects and tools to collect and measure system metrics to automate the appropriate resizing work to meet the required load while saving resources to optimize operating costs.

The scalability of each Serverless computing platform is different. Both OpenFaaS and Knative have the ability to automatically change the size of the service function based on the amount of load required to process or can be based on the resources needed to process requests such as CPU, RAM. However, the ability to automatically adapt based on the resources needed to process the request depends on a controller of the K8S platform, which is the controller that changes the number of Pods vertically. This controller will monitor the resource consumption parameters of the Pods containing the logic of the execution function such as CPU or RAM. If the consumption of these resources exceeds the pre-configured limit, the controller will send a request to change the number of Pods running the function and redirect the request to a new Pod for processing. In the scope of this project, we will try to measure the ability to automatically expand and resize of the Serverless computing model, and thus we will only consider input request load to the platform. Base on its ability to automatically expand and resize in accordance of the growing of the demand load into the system, OpenFaaS will refer to the parameter of the average request rate sent to the OpenFaaS Gateway, while Knative will refer to the parameter showing the number of requests sent to an entity containing the processing function at the same time. The average request rate may be higher or lower than the number of concurrent requests sent to the function depending on whether the time spent processing a function call is high or low respectively.

In this work, we aim to evaluate and compare the scalability of OpenFaaS and Knative, we will measure the responsiveness of these two platforms with different request load scenarios. To ensure objectivity and fairness, we will choose the function call elements to be completely identical and the configuration to automatically change the number of functions is also the same. We choose to use a common function containing the logic to process incoming requests. To simplify and suit the testing context in this project, we choose a function that prints the content of the incoming request ("echo" function in Linux operating system). For example, if the incoming request has the content "Hello-world", the function will process this request and return the result on the terminal as "Hello-world". The source code of the function is written in the Golang and function calls are sent via the HTTP with the POST method. The HTTP request generator and function request dispatcher chosen is "**hey**" which is an open source tool used to measure the performance of systems serving requests over the HTTP. The request dispatch scenario is as follows: Metrics such as request thread time sent to the function, number of concurrent simulated request dispatchers, maximum number of concurrent requests, request content are exactly the same on both platforms. The internal configurations of OpenFaaS (Fig. 5) and Knative (Fig. 6) are set with the same configuration parameters such as minimum and maximum number of function running entities, computing resources provided to each entity, number of concurrently increased or decreased entities. In addition, the request dispatch rate limit of OpenFaaS is set to 10 requests and the request dispatch limit to Knative's concurrent function processing entity is also set to 10 requests.

In the Knative platform, the main component that control the process of auto-scaling or rollback is the Autoscaler. The Autoscaler is responsible for adjusting the maximum scaling rate and settings related to the smallest and largest number of Pods on a given revision. The maximum scaling rate is the rate at which the Autoscaler allows the current number of instances to increase or decrease during each scaling trigger. Additionally, the Autoscaler decides when incoming requests should be routed to the Pod running the function instead of to the Activator, which makes sense in scenarios where the Pod count is increased from 0 or reduced to 0. Requests cached by the Activator will send a trigger signal to the Autoscaler, which will then update the number of Pod replicas specified in the Deployment of a given revision.

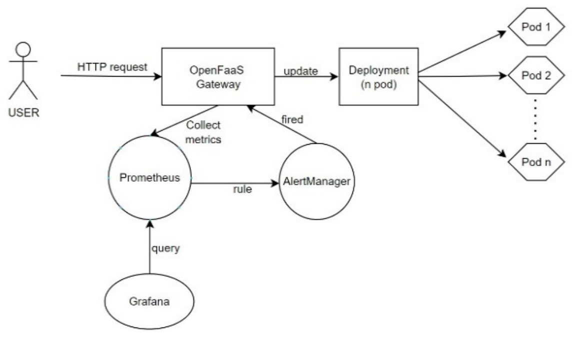

Fig. 5. Operational flow during OpenFaaS auto-scaling or rollback

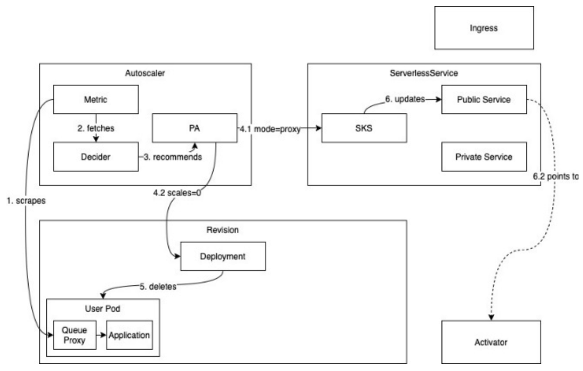

Fig. 7. The scenario on which the platform to recall all Pods to zero

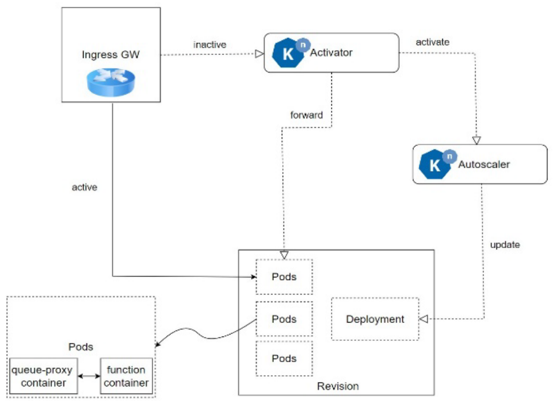

Fig. 6. Key components involved in implementing the auto-scaling or rollback of Knative

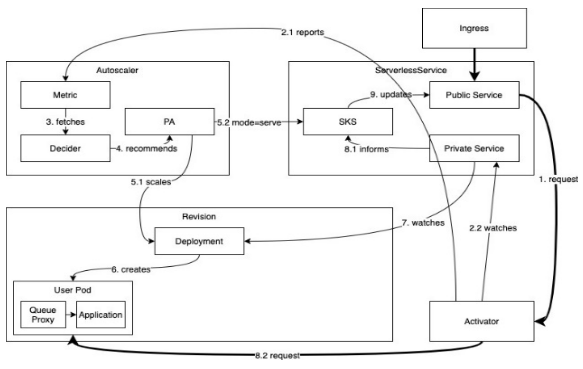

Fig. 8. The scenario on which the platform to initiate new Pods from zero

When the system is running in a stable state, the Autoscaler continuously scans the currently running instance Pods to adjust the scale of the instance continuously. The Autoscaler will monitor the values collected from the Pods to decide whether to scale up/down the instance. As requests come into the system, the collected values will change and the Autoscaler will request the deployment of the instance to follow a specific number of Pod replicas. This means that the Autoscaler will adjust the number of Pods running the function that exists in the current instance to meet the increasing request load, ensuring that the system is always stable and follows the defined scale. Next, the Scalable Kubernetes Service SKS component continuously monitors changes in the scale of the instance through a private Service, from which it updates the public Service accordingly.

In the scenario that need to rollback or initiate new Pod the Autoscaler and Activator components will process through several steps as depicted on Fig. 7 and Fig. 8 respectively.

In this section, we already clarify the methodology for evaluating the scalability and performance of Serverless computing platforms by outlining a step-by-step process for building a real-world test model for the OpenFaaS and Knative platforms. Next section will focus on analyzing the flow and operations of the components inside the OpenFaaS and Knative computing architectures to administrating the platform scalability and performance to meet the changing needs of users.

## 4. Testbed and Performance Evaluation

Follow the proposed methodology on section 3, we have designed and implemented the Test-bed to run and to measure the scalability and performance KPIs of the open source Serverless computing architectures. The Test-bed building is illustrated on the Fig. 9.

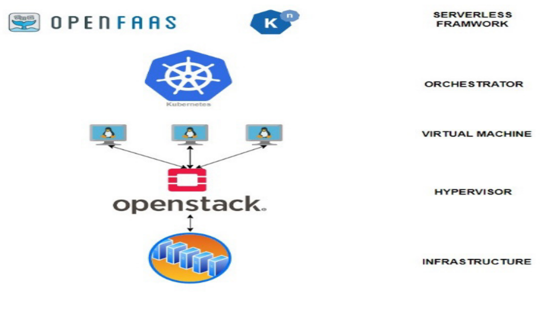

Fig. 9. The Test-bed model

The image above shows the overall architecture and technologies we use to implement the Serverless computing platform in the experiment. To provide computing resources (CPU, RAM) and storage resources (Storage) for the above layers, we use a physical server hardware infrastructure at the FPT Cloud data center. The server virtualization tool is OpenStack to create virtual machines that will run the Kubernetes components. The K8S cluster is built from 3 virtual machines including 1 management machine (master node) and 2 servers (worker nodes) configured according to the parameters below:

- *Master node: 2 core CPU, 4 GB RAM, 40GB SSD. Ubuntu OS 20.04.4 LTS*
- *Worker node: 2 core CPU, 4 GB RAM, 100GB SSD. Ubuntu OS 20.04.4 LTS*

After setting up the K8s cluster, we have deployed the OpenFaaS and Knative platforms on it using Helm tool, as illustrated on the Fig. 10 and Fig. 11 respectively.

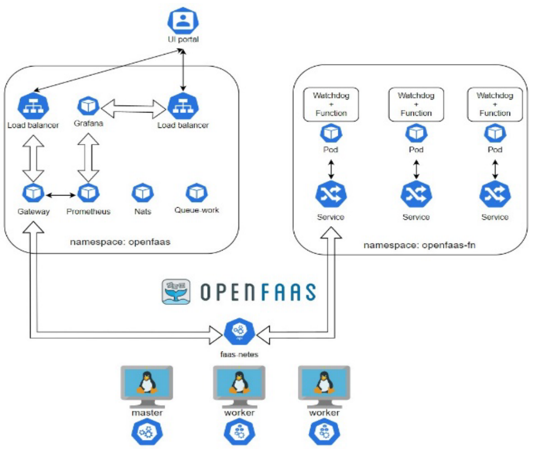

Fig. 10. Deployment diagram of system components of OpenFaaS platform

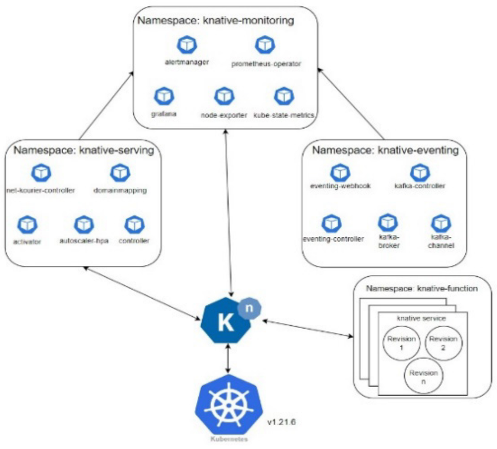

Fig. 11. Deployment diagram of system components of OpenFaaS platform

The major testing results for several scenarios are shown following.

### *4.1 Processing Performance without Pod Running the Function*

The ***go-echo*** function has been deployed to both Serverless computing platforms. According to the configuration presented above, this function allows all Pods to be reclaimed to 0, which means that when no requests are sent after a period of idle time, the function won't be instantiated in the Pods. When the function is newly initialized and has not received any processing requests, it will not create a Pod to run the function. At this time, there are no active Pods on the OpenFaaS-fn and Knative-fn namespaces. We will then send a processing request to the ***go-echo*** function on OpenFaaS and Knative to compare the successful response time and how long it takes when there are no Pods running the function. The tool used to send HTTP requests to call the function is **Postman** with the POST method through the URL provided by OpenFaaS and Knative when creating the function.

Table 1. Comparison of the time to successfully process the request when no Pod is running the function

| Function call No. | OpenFaaS (s) | Knative (s) |
|---|---|---|
| 1 | 6.23 | 2.33 |
| 2 | 6.12 | 2.55 |
| 3 | 6.27 | 2.21 |
| 4 | 6.09 | 2.48 |
| 5 | 6.21 | 2.12 |
| 6 | 6.12 | 2.69 |
| 7 | 6.05 | 2.21 |
| 8 | 6.34 | 2.88 |
| 9 | 6.12 | 2.57 |
| 10 | 6.05 | 2.44 |
| Average | 6.16 | 2.45 |

Under exactly the same conditions, with the same request content, we examined 10 function calls to OpenFaaS and Knative when there was no Pod running the function. From Table 1, it can be seen that the request processing time under the condition of no Pod running the function from the Knative platform is much better. The processing time and response time for OpenFaaS requests is nearly 3 times larger than that of Knative. The reason why OpenFaaS's request processing time is longer is because of the initialization time and configuration settings for the Pod running the function to process OpenFaaS requests take more time. The initialization process of the first Pod running the function in the Knative platform is completed faster because the architecture of this platform takes advantage of many coordination APIs provided by K8S, the Pod creation process in Knative is more optimized. The above survey shows that the feature of automatically reclaiming all Pods to 0 on Knative has better performance, requests sent to functions when there is no Pod running the active function are responded faster on Knative, which contributes to improving the quality of user experience.

### *4.2 Elasticity in the Case of Larger Traffic Demand*

To simulate the actual operational environment, I chose the open-source traffic generation tool ***hey*** [16]. ***Hey*** allows generating traffic loads and testing the performance KPIs for web applications. It supports sending a lot of concurrent HTTP requests and collecting back information about response time, statistics and errors from the server. I use ***hey*** to fire requests to functions in both OpenFaaS and Knative in a period of 5 minutes and the number of concurrent connections sending requests are 100 connections.

The Fig. 12 shows the process of continuously sending traffic load to the system for intervals of 5 minutes, the number of Pods increases steadily and reaches a maximum of 8 Pods before we stop sending the traffic. The input throughput through the Gateway is stable at more than 1000 requests per second. It is found that increasing the number of Pods does not help improve the throughput through the Gateway, the number of Pods running the function increases slowly but does not affect the ability to receive requests, does not cause congestion at the Gateway. When the Gateway does not receive any more requests, OpenFaaS immediately reclaims the resources of the Pods running the function and brings the number of Pods to 0 to save resources.

The test scenario is similar to OpenFaaS, but the function in Knative changes the number of Pods running the function continuously. Fig. 13 shows that Knative immediately creates as many Pods as possible to handle the initial incoming requests. After the Pods are successfully initialized and participate in handling the requests, the number of simultaneous requests on each Pod decreases, Knative proceeds to collect the actual parameters to update the number of Pods running the function. Compared to OpenFaaS, Knative is more sensitive to the input load, the number of Pods is changed in a short period of time, when the Pod is revoked to 0, Knative will also gradually reduce the number of Pods to 1 Pod and then wait a short time before revoking all Pods to 0 to ensure that no more requests are sent to the function. After this comparison scenario, we can see that Knative's ability to adapt to input load is better than OpenFaaS, the number of Pods changes continuously to ensure that Pods do not have to handle a number of simultaneous requests exceeding the set threshold. However, OpenFaaS's stability is higher than Knative, this platform will expand the number of Pods only when really necessary and only increase 1 Pod after each change, this will not take up too much of the system's computing resources, causing conflicts and resource contention with other functions or being deployed.

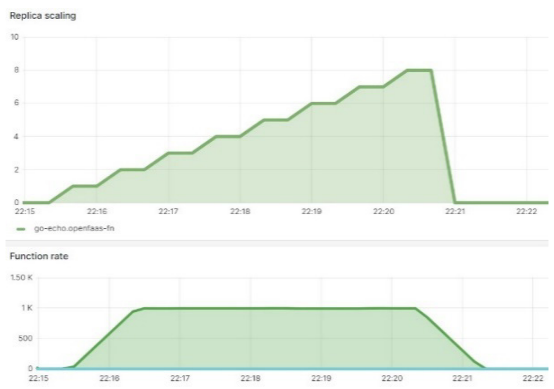

Fig. 12. OpenFaaS's process of automatically changing the number of Pods running functions according to the input traffic load

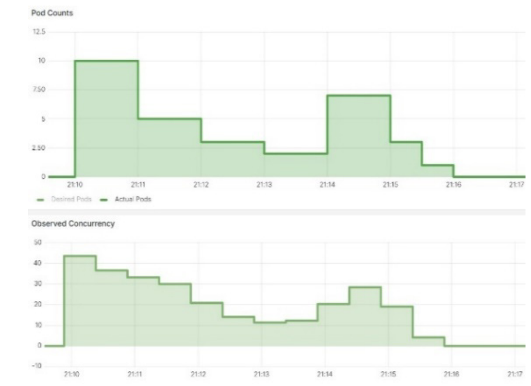

Fig. 13. Knative's process of automatically changing the number of Pods running functions according to the input traffic load

Even though on this testing we focus on measuring of RAM/CPU usages and number of Pods needed upon traffic to the Serverless platform increased, however some other performance metrics may also important like the cold start times. When our servers don't run all the time in the Serverless setup, they have a cold start time associated that is required by the infrastructure to initialize resources and boot up instances when the customer or client request arrives or an event occurs. This boot up time is not good for latency-sensitive use cases and is what developers contemplate when considering Serverless model for their service architectures. But that might not be an issue for services that are not latency-sensitive and want to leverage the upsides of Serverless architecture.

Not all Serverless implementations have the same cold start time. Factors that influences it are the choice of runtime, configuration settings and whether the function is a part of a virtual private cloud or not. While some cold starts may take a few seconds, others can be much quicker. On future paper, we may setup and conduct the tests to measure this metric under various scenarios and configurations but for now, this paper only focuses on measuring the scalability of the platform other than performance detailed metrics.

## 5. Conclusion

Applications built on Micro-service architecture are increasingly popular and effective. Serverless cloud computing platforms have emerged to support the deployment and operation of these applications by dividing the source code into small independent units called functions and deploying them on Serverless computing platforms. Open source serverless computing platforms are developed by the community to create a FaaS model. The strengths of these platforms are a large support community, integration with many infrastructures and the ability to expand and customize the source code freely without depending on the supplier. The content of this paper provides the main concepts of cloud computing service provision models and outlines the advantages and disadvantages of the Serverless computing model. We have designed the methodology and setup the theoretical basis to analyse and to measure the scalability and performance KPIs of the open source Serverless computing platforms. Finally, we successfully designed and built a real test model for the two most popular Serverless platform, namely OpenFaaS and Knative, created a test scenario to compare and evaluate the elasticity of the platform in condition of high traffic load input and measured the processing performance of the function. The overall evaluation results show that the function processing performance before the Pod running the function of Knative is 3 times faster than OpenFaas. Knative platform scales more flexibly with load demands and performs better in handling function requests than OpenFaaS platform under the same test conditions.

Recognizing that the request processing performance of OpenFaaS is limited. The reasons are the facts that requests going through the Gateway are not routed to the correct Pods with enough computing resources (the Gateway does not have the ability to balance the load). That's why for future research, we will propose a solution to create a Load Balancer Service integrated into the OpenFaaS Gateway component to provide the ability to coordinate the appropriate load to the Pods running the function. In addition, it is necessary to design more copies for the OpenFaaS Gateway component to increase the load capacity, avoiding the situation where incoming requests are congested here, causing the request flow to be interrupted.

## References

[1] Cloud team. PaaS vs IaaS vs SaaS, IBM. [Online]. Available: https://www.ibm.com/topics/iaas-paas-saas, Accessed on: June, 2023.

[2] Development center. What is Serverless, Redhat. [Online]. Available: https://www.redhat.com/en/topics/cloud-native-apps/what-is-Serverless, Accessed on: April 2023.

[3] J. Li, S. G. Kulkarni, K. Ramakrishnan, and D. Li, Understanding open source Serverless platforms: Design considerations and performance, Proceedings of the 5th international workshop on Serverless computing, 2019, pp. 37-42. https://doi.org/10.1145/3366623.3368139

[4] Documentation team. Knative docs, Knative Forum. [Online]. Available: https://knative.dev/docs, Accessed on: August. 2024.

[5] Documentation team. OpenFaaS docs, OpenFaaS forum. [Online]. Available: https://docs.openfaas.com, Accessed on: Feb. 2024.

[6] Documentation team. OpenWhisk docs. [Online]. Available: https://github.com/apache/openwhisk, Accessed on: Sept. 2024

[7] S. K. Mohanty, G. Premsankar, M. Di Francesco *et al.*, An evaluation of open source Serverless computing frameworks, 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Nicosia, Cyprus, Dec. 10-13, 2018. https://doi.org/10.1109/CloudCom2018.2018.00033

[8] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, Serverless computation with openlambda, in Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing, ser. HotCloud'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 33-39.

[9] R. Vojta, AWS journey: API Gateway & Lambda & VPC performance. [Online]. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/rest-api-optimize.html, Accessed on June. 2023.

[10] E. Jonas, Microservices and Teraflops. 2016, [Online]. Available: http://ericjonas.com/pywren.html, Accessed on May. 2023.

[11] E. d. Lara, C. S. Gomes, S. Langridge, S. H. Mortazavi, and M. Roodi, Poster abstract: Hierarchical serverless computing for the mobile edge, 2016 IEEE/ACM Symposium on Edge Computing (SEC), Oct. 27-28, 2016. https://doi.org/10.1109/SEC.2016.37

[12] Amazon Web Services, AWS Lambda@Edge, 2017 [Online]. Available: http://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html. Accessed on: June. 2024

[13] AWS Greengrass, 2017 [Online]. Available: https://aws.amazon.com/greengrass, Accessed on: Feb. 2023.

[14] Documentation team. Knative community. [Online]. Available: https://github.com/knative, Accessed on: March. 2024.

[15] Stefanprodan. Grafana for FaaS. [Online]. Available: https://github.com/stefanprodan/faas-grafana, Accessed on: Feb. 2023.

[16] Documentation team. Hey - an HTTP traffic generation tool. [Online] Available: https://github.com/rakyll/hey, Accessed on: May. 2023