EasyChair Preprint

№ 1735

Pigeon: A Dynamic and Efficient Serverless and FaaS Framework for Private Cloud

Wei Ling, Lin Ma and Chen Tian

October 21, 2019

Pigeon: A Dynamic and Efficient Serverless and FaaS Framework for Private Cloud

Wei Ling
Futurewei Technologies.
Santa Clara, USA
weiling6@gmail.com

Lin Ma
Futurewei Technologies.
Santa Clara, USA
lin.ma@futurewei.com

Chen Tian
Futurewei Technologies.
Santa Clara, USA
chen.tian@futurewei.com

***Abstract*—Recently, voice-triggered small cloud functions such as Alexa skills [1], and cloud mini programs [2] for IoT and smartphone, grow exponentially. These new developments also attract organizations to host their own cloud functions or mini programs in private cloud environment and move from traditional Micro-service architecture to Serverless Function-as-a-Service (FaaS) architecture. However, current Serverless FaaS frameworks [3] [4] [5] [6] [7] cannot meet cold start latency, resource efficiency required by short-lived cloud functions and mini programs.**

In this paper, we build a new Framework — Pigeon that brings Serverless and FaaS programming paradigm into private cloud to enable enterprises to host these applications. Pigeon creates function-oriented Serverless framework by introducing an independent and finer-grained function-level resource scheduler on top of Kubernetes. A new oversubscription-based static pre-warmed container solution is also proposed to effectively reduce function startup latency and increase resource recycling speed for short-lived cloud functions. Empirical results show that Pigeon framework enhances function cold trigger rate by 26% to 80% comparing to AWS Lambda Serverless platform [5]. Comparing to Kubernetes native scheduler based serverless platforms, throughput gets 3 times improvement while handling short-lived functions.

***Index Terms*—Serverless, FaaS, Dynamic Resource Management, Private Cloud**

I. Introduction

While Public cloud serverless platforms, such as AWS Lambda [5], Microsoft Azure Function [8], and Google Cloud Function [9], show advantages, such as flexibility of scaling, speed of development and cost reduction, etc., the downsides of serverless solutions are obvious and remain unsolved. The top 3 downsides of serverless approach are **portability**, **control**, and **performance** [10] [11]. First of all, public cloud serverless approach requires enterprises lock-in to one cloud vendor, and reduces software portability. Secondly, for some cloud applications, the serverless pay-per-use model reduces cost. But for many large-scale or computational-intensive applications, the control for both cost and scheduling becomes very challenging [12] [13]. Thirdly, application performance, especially cold startup latency, is not guaranteed by any of the public serverless platforms. Therefore a better cloud serverless solution is needed to overcome these issues.

Together with serverless, Function-as-a-Service (FaaS) is attracting prevailing attention recently with the emerging of voice-triggered skills, action functions [1] [14], and cloud mini program [2]. It fundamentally changes the way developers build applications [10]. FaaS enables small programs, i.e. stateless functions, to run on cloud platform with trigger action based execution model. (Trigger refers to function invocation request and action means function being executed.) Unlike traditional long-lasting micro-service, the skill function is small in size but large in quantities. For example, the amount of cloud skill functions developed by community for AWS Alexa reaches 80,000 [1]. These skill functions are replacing smart phone APPs with increased varieties and enhanced fast responsiveness.

In private cloud FaaS approach, an organization rents or purchases a set of cloud servers, such as AWS EC2. Using FaaS and serverless programming model and execution framework, developers can develop and execute their code the same way as for Lambda functions on public serverless cloud. It enables quick serving small program as function with small footprint and simple life cycle management, and expands applications that use cloud as a platform.

II. Motivation

The main motivation of building Pigeon — a highly dynamic private cloud serverless and FaaS framework is to host tiny cloud functions of large quantity and overcome drawbacks that public serverless infrastructure suffers. First of all, FaaS function cold startup latency is always a major concern for serverless approach. The time it takes to start Docker container, download function and initialize running environment may exceed 3$\sim$100 sec [15], [16], which limits FaaS use-cases to only those slow-responsive applications. For the mini programs and skill functions, they are started occasionally when there is a trigger. The cold startup latency has direct impact on user experience. Second, Kubernetes [17] is becoming a popular container orchestration platform that supports most open source serverless FaaS frameworks [4], [3], [18], [7], [6]. However, scheduling and resource management of Kubernetes are designed exclusively for long running micro-services. They cannot meet FaaS' short-lived, fast resource reusing requirements in a private cloud environment. An additional finer-grained function-level resource management scheme is needed on top of Kubernetes. Third, cost control is another important factor to choose private cloud implementation. Pay-per-use model may be affordable for

individual service providers. But for large FaaS infrastructure providers who allow their own customers to write and run FaaS programs, the public serverless billing model gives little control on overall budgeting and spending.

III. Kubernetes Based FaaS Framework Survey

A. FaaS Function Execution Model

Popular open source Serverless Frameworks, such as OpenWhisk [4], OpenFaaS [18], Kubeless [6], Fission [3], and Nuclio [7], are all designed to run on top of Kubernetes. This makes these serverless frameworks portable to different cloud providers' Kubernetes-based cloud environments, e.g. Google GKE [19], AWS EKS [20], and Azure AKS [21], which solves vendor lock-in problem mentioned in Section I. The function execution process of OpenWhisk [4] is based on spins up a Kubernetes-managed docker container from a pool that acts as execution unit for a chosen function. Function trigger and execution are handled by Kubernetes. Fission [3], instead, maintains a pool of generic environment Kubernetes pods. Once a function is invoked, one of pods from the pool is taken and used for execution. OpenFaaS [18] and Nuclio [7] work natively with Kubernetes. Each function user builds creates a Docker image. which when deployed, creates a Kubernetes service and that in turn creates a number of Pods for execution. Kuberless [6] uses Kubernetes Customer Resource Definition (CRD) to represent function. A function custom object will be created and managed by independent CRD controller. The function is exposed as a Kubernetes service. As we can see, the function execution model of these open source serverless frameworks all rely on Kubernetes' service, pod, main controller, and resource scheduler. While it simplifies overall design, but it incurs all limitation imposed by Kubernetes, e.g. high startup latency and slow resource recycling speed.

B. Pre-warmed Container Pool Approach

FaaS Function cold startup latency and resource recycling latency are major concerns for short-lived skill functions and mini programs running on top of Kubernetes. Kubernetes resource recycling latency is mainly due to heavy interactions among Kubernetes API servers, distributed Kubelets and multiple Kubernetes managers. The Function cold startup latency depends on container type, container image size, programming language runtime [22], function and its dependent package size, network bandwidth, memory cache, and local volume settings [11]. If all these components and packages are freshly downloaded from database and initialized on demand, it may take more than 130 sec [15]. In order to reduce the cold start latency, pre-warmed containers are adapted by OpenWhisk [4] and Fission [3]. Openwhisk [22] keeps 3 container pools, e.g. hot container pool, warm container pool and pre-warmed container pool. The pre-warmed containers are pre-started and loaded with common running environment and language runtime but without function and its dependent package. The warm and hot container have user function and specific dependent package loaded. The hot container is in active state and the warm container is in sleep state. Openwhisk relies on its Invoker to make the decision of either reusing an existing hot container, or resuming a paused warm container, or start to use a pre-warmed container, or launching a new 'cold' container and transforming it to pre-warmed containers. Invoker also monitors the size and health of the pools. Fission [3], on the other hand, maintains a pod pool of pre-warmed containers to host function. The pool size of initial pre-warmed containers can be configured based on user needs and managed by the pool manager. Pigeon framework also uses a pre-warmed container pool. In the following sections, we will discuss how Pigeon enhance performance and dynamics of its pre-warmed container pool.

C. Pre-warmed Container Pool Management

While pre-warmed container pool reduces FaaS function startup latency, managing the pool brings new challenges. For example, different resource-sized containers are needed in the pool for diversified functions. The resource includes CPU, memory or GPU, etc. Traditionally, container is managed by Kubernetes controller. Once a container is created, the resources that required by the container are actually reserved and cannot be modified on the fly. Therefore, a combination of different sized containers needs to be carefully planned for the pool ahead of time. As we cannot predict incoming trigger patterns and function footprints, resource segmentation is inevitable, i.e. some smaller-sized containers may not be used at all if all incoming function are large, or some large-sized container may end up running smaller functions if smaller containers are all taken.

The pre-warmed container pool solution also comes with a dynamic container creation scheme to maintain the pool size [3] [23]. In principle, once a container is assigned to a new function, a same-sized empty container will be created immediately. The pre-warmed container pool size is of key importance. If the pool size is not big enough, in case of burst requests, entire pre-warmed container pool may be drained out and function triggers have to wait for new pre-warmed containers to be created from cold. This situation is very common in private cloud environment since overall resources are quite limited and the pool size is oftentimes moderate. Increasing pre-warmed container pool size is not a good choice either. Since pre-warmed container combination in the pool cannot fully match with the unpredicted incoming function requests, the number of un-used or under-used containers will be large and the overall resource utilization may be low.

IV. Highly Dynamic Pigeon Framework

A. Oversubscribed Static Pre-warmed Container Pool

One novelty of Pigeon framework is to use an oversubscribed static pool that contains pre-warmed containers with all combinations of resource sizes. Each FaaS service pod is preloaded with this static pool. In Pigeon, one FaaS service maps to only one FaaS service pod due to intra-service load-balancing restrictions. The function-level resource scheduler ensures only some of these pre-warmed containers of a FaaS

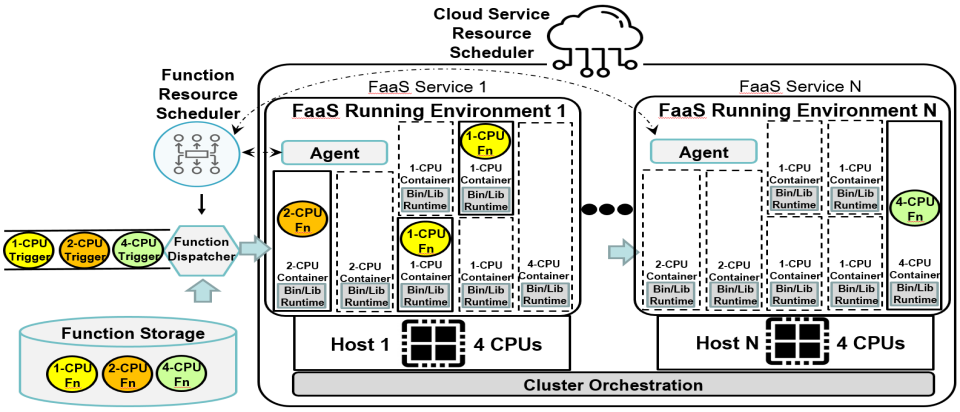

Fig. 1: Two-level Resource Management: a function-level resource scheduler on top of Kubernetes service-level resource scheduler.

service are assigned functions and overall resource usage is always in check and never exceeds upper limit. Figure 1 illustrates an example of a set of FaaS services. The overall resource request of each FaaS service is 4-CPU, thus the Pod of the FaaS service is assigned to 4-CPU host by Kubernetes. In the FaaS service 1, seven pre-warmed containers are preloaded. Among them, four containers have 1-CPU resource upper limit, two containers have 2-CPU resource upper limit, and one container has 4-CPU resource upper limit. This kind of pre-warmed container size partition can serve nearly all kind of function triggers with different function size requests. For example, it can serve either four 1-CPU functions, or two 2-CPU functions, or a single 4-CPU function, or any combination of them. Even though these pre-warmed containers are started with preloaded language running environment and runtime, they only consume less than 1% CPU since no function is really loaded in it. A more comprehensive resource partition scheme is defined in next sub-section to cover all different kind of resource types, including GPU, CPU and memory.

B. Two-level Resource Management

Another novelty of this proposed framework is to introduce an independent function-level resource management scheme on top of Kubernetes service-level resource scheduler. The function-level resource scheduler is used to guide function dispatcher to dispatch function trigger towards a FaaS service with sufficient resource. Agent in each FaaS service pod then finds a proper-sized container serving the function trigger. The Kubernetes resource scheduler on the other hand is only responsible for creating, deploying, and scaling FaaS services and pods. Figure 1 shows an example of how the two-level resource management scheme works. There are N FaaS services and pods deployed on N 4-CPU hosts by Kubernetes resource scheduler. The agent in each FaaS service pod is responsible for syncing up available resources with the function resource scheduler. The function resource scheduler is aware of all available resources in each FaaS service in a cluster, guides function dispatcher to dispatch the first function trigger for a function to a proper FaaS service, and updates remaining resources of the FaaS service. The function dispatcher caches the first triggered function and FaaS service mapping, and dispatches the subsequent function triggers of the same function directly without getting function resource scheduler involved again.

Figure 1 also gives an example of how the functions are assigned to pre-warmed containers after trigger reaches service pod. The agent assigns two 1-CPU functions to two 1-CPU pre-warmed containers, and one 2-CPU function to a 2-CPU pre-warmed container in FaaS service pod 1. In FaaS service N, one 4-CPU function is loaded to a 4-CPU pre-warmed container to have saturated the host resource. The function runtime inside pre-warmed container handles function requests and loads corresponding function from function storage.

C. Pigeon Function Level Resource Management

1) Function resource presentation:

We define resource subscription vector for each container as $X_j = \{x_{1,j}, x_{2,j}, x_{i,j}, \ldots, x_{n,j}\}$ where j is container index, i is resource type index, n is number of resource types, and $x_{i,j}$ refers to the subscription upper limit of resource type i on container j. Table I shows a list of containers in FaaS service k and their upper limit resource subscription breakdown. Note that $X_{j,k}$ represents the resource subscription vector for container j in FaaS service k.

TABLE I: Container Resource Subscription Vector Example

Resource Container	Resource Vector	Resource Types ($i \in [1..n], n = 3$)		
		CPU	GPU	Memory(GB)
1	$X_{1,k}$	$x_{1,1} = 1$	$x_{2,1} = 0$	$x_{3,1} = 1$
2	$X_{2,k}$	$x_{1,2} = 2$	$x_{2,2} = 0$	$x_{3,2} = 1$
3	$X_{3,k}$	$x_{1,3} = 4$	$x_{2,3} = 0$	$x_{3,3} = 2$
4	$X_{4,k}$	$x_{1,4} = 6$	$x_{2,4} = 1$	$x_{3,4} = 4$
5	$X_{5,k}$	$x_{1,5} = 1$	$x_{2,5} = 1$	$x_{3,5} = 2$

We also define $Y_k = \{X_{1,k}, X_{2,k}, X_{j,k}, \ldots, X_{m,k}\}$ as an exhaustive vector of pre-warmed containers in FaaS service Y_k, where m is the total number of pre-warmed containers in service k. m is set to be bigger than the containers number that will be actually used to enable over-subscription and facilitate function assignment. The vector Y_k will not be changed

once FaaS service k is created. $A_k = \{x_{1,k}, x_{2,k}, \ldots, x_{n,k}\}$ represents available resources in FaaS service k. The vector of available resource for each service is used for function scheduler to determine whether a new function trigger should be allowed to dispatch to a FaaS service.

$Y = \{Y_1, Y_2, Y_k, \ldots, Y_l\}$ is a vector of all created FaaS services in a cluster (cloud). This vector can be scaled up or down by cloud service scheduler. Table II shows some FaaS services in a cluster and corresponding available resources.

TABLE II: Available Resource A_k in a Service Y_k

FaaS	Available	Resource Types ($i \in [1..n], n = 3$)		
Service	Resource	CPU	GPU	Memory(GB)
Y_1	A_1	$x_{1,1} = 1$	$x_{2,1} = 1$	$x_{3,1} = 0.5$
Y_2	A_2	$x_{1,2} = 2$	$x_{2,2} = 1$	$x_{3,2} = 0.9$
Y_3	A_3	$x_{1,3} = 1$	$x_{2,3} = 2$	$x_{3,3} = 1$
Y_4	A_4	$x_{1,4} = 1$	$x_{2,4} = 0$	$x_{3,4} = 0.5$
Y_k	A_k	$x_{1,k} = 6$	$x_{2,k} = 1$	$x_{3,k} = 8$

2) Function resource management algorithm:
A function needs to be dispatched to a FaaS service and then a pre-warmed container. We formulate resource request of a function trigger q as $F_q = \{x_{1,q}, x_{2,q}, \ldots, x_{n,q}\}$. The scheduler will pick the first service that satisfies $A_k \geq F_q$. Specifically, available resource needs to be more than the requested resource for each of the n resource types. Agent in the selected FaaS service pod picks a right sized pre-warmed container, e.g. $X_{j,k} = F_q$, for loading the function and processing the trigger.

The available resource A_k in FaaS service k will get decreased by function resource scheduler when the function dispatcher sends a new function trigger to the service. Agent in each FaaS service pod will increase the available resource when a function lifecycle ends. After available resources are changed in a FaaS service, a list of valid containers will be recalculated for future function assignment. For example, if some resources are used in a FaaS service, some large sized containers may not be valid and should be moved out of the valid list. It uses A_k to measure if containers X_j in a service k can still obtain its targeted resources $X_j = \{x_{1,j}, x_{2,j}, x_{i,j}, \ldots, x_{n,j}\}$ based on current resource availability A_k in service k. If any type of resource becomes insufficient, this pre-warmed container becomes invalid. A new valid pre-warmed container list in service Y_k is regenerated.

Table III shows a new valid pre-warmed container list in service k after resource are consumed by a function F_q={4, 0, 2} taking 4 CPUs, 0 GPU, and 2G Memory. Comparing to Table I, only 3 pre-warmed containers are available instead of 5 before.

TABLE III: New Available Containers in Service k

Container	Vector	CPU	GPU	Memory(GB)
1	$X_{1,k}$	$x_{1,1} = 1$	$x_{2,1} = 0$	$x_{3,1} = 1$
2	$X_{2,k}$	$x_{1,2} = 2$	$x_{2,2} = 0$	$x_{3,2} = 1$
5	$X_{5,k}$	$x_{1,5} = 1$	$x_{2,5} = 1$	$x_{3,5} = 2$

V. Pigeon Implementation

The implementation of Pigeon framework consists of 7 major modules including Kubernetes, FaaS Controller, FaaS Data Plane, Function Integrated Development Environment(IDE)/Command Line Interface(CLI), FaaS Service Pod, Function/Docker Storage and Kubernetes Etcd. (Figure 2).

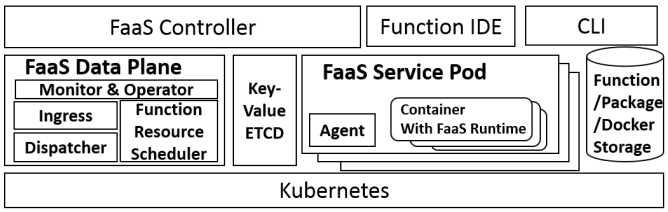

Fig. 2: Pigeon Framework Architecture

- **Kubernetes:** Foundation of Pigeon framework that handles cloud orchestration, resource scheduling, and scalability management of all Pigeon components.
- **FaaS Controller:** it deploys and manages FaaS data plane components using operator.
- **FaaS Service Pod:** FaaS service pod hosts execution of FaaS functions. It consists of an agent and multiple hot and pre-warmed containers with FaaS runtime running in it. The agent also monitors health of each container and recycles the container when function lifecycle ends.
- **FaaS Data Plane:** All components of this module are implemented as Kubernetes services. The monitor and operator component is used to monitor the health of each component and deploy a new instance if needed. Ingress service handles incoming restful function trigger and translates it to internal function trigger. Function dispatcher service distributes function triggers to a FaaS service, either through cached function to FaaS service mapping, or through Function resource scheduler. Function resource scheduler service monitors available resources of FaaS services, and picks a suitable FaaS service for sending a function trigger.
- **Storage:** We choose object store database for FaaS Function/Package/Docker storage and use Kubernetes Etcd for resource counting.
- **Function IDE/CLI:** Function editing relies on Function IDE. The CLI module supports FaaS function creation through SAM [24] compatible 3rd party IDE.

VI. Experiments and Evaluation

Extensive experiments are conducted to compare the efficiency and performance of Pigeon framework with other Kubernetes based serverless frameworks.

A. Function First Trigger Latency Comparison

Function first time trigger round trip latency is considered as a key measurement of user experience for skill functions and mini program. Experiment is performed on a private cloud of 6 nodes, each with 32 cores and 125GB memory. The FaaS Functions used in these experiments are Node.js empty FaaS functions that directly respond to restful API trigger.

The function image together with its dependent libraries are packaged as a tar file and will be downloaded from database to runtime after corresponding trigger arrives. The tar file size is 100MB. Skill function CPU usage is set to 25% CPU. We send 200 first time triggers of 200 different FaaS functions to the cluster the same time and measure overall response time. We obtain function first trigger rate (function/sec) by dividing 200 functions by overall response time.

For Pigeon framework, 100 over-subscribed multiple-sized pre-warm container pools are pre-created on each node, the actual CPU and memory utilization of these pre-warmed containers per node are negligibly about 1% and 80MB respectively. A totally 600 over-subscribed pre-warmed containers are created for the private cloud cluster under testing.

For dynamic pre-warmed pool solution and Fission framework [3], pool size used in the test varies from 40 containers to 200 containers per cluster. They are tested in the same private cloud. The dynamic pre-warmed pool solution is used to mimic the pool management of OpenWhisk [4] [23] and used as a reference.

Figure 3 shows that the over-subscribed static pool solution used in Pigeon framework supports a constant first time trigger rate of about 60 function/sec. That of the dynamic pre-warmed container pool solution varies from 2.1 function/sec, 2.5 function/sec, 5 function/sec, and 18 function/sec corresponding to initial pool size of 40, 80, 160, and 200 containers respectively. The bigger pool size, the better first time trigger rate. This is because, when first time trigger injection rate is high, dynamic pool solution may saturate all its standby pre-warmed containers thus other triggers cannot be served until new pre-warmed container and its corresponding service/pod is created by Kubernetes native scheduler. Therefore, bigger pool size can reduce first trigger latency. However, bigger dynamic pool size causes lower resource utilization as depicted in section 3.3. A win-win solution does not exist in dynamic pool approach. On the other hand, the over-subscribed static pool solution used in Pigeon framework has enough right-sized pre-warmed containers, provides constant and 3 to 10 times higher performance than dynamic pool approach.

We also create the same 200 empty Node.js functions on AWS Lambda platform and run the same tests. We measure the 200 function first trigger rate and got the inconsistent results. It varies from 12 functions/sec to 44 functions/sec, 80% to 27% lower than what Pigeon offers. This is a known issue of public cloud serverless solution. The first time trigger rate of Fission is about 1.26 function/sec. The reason the first time function startup rate is so low is that, even though Fission claims using a pre-warmed container pool, a new Kubernetes service and function-to-service mapping needs to be created at function first loading time, which cancel all gains from pre-warmed container approach.

B. Cloud Resource Efficiency Comparison

In private cloud environment, cloud resources are limited. In order to support large quantity of small functions, it requires short function life span and fast container/cloud resource

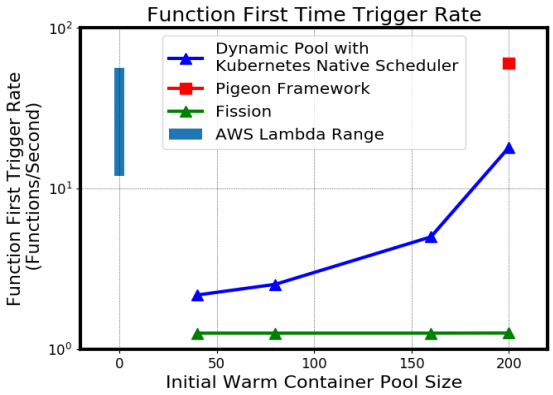

Fig. 3: FaaS Function Startup Rate

recycling. In this test, we create 400 empty Node.js FaaS functions that requires 50% CPU upper limit each and 200 CPU resources in total. The cluster has only 100 CPU resources and can run 200 functions simultaneously. In order to serve all 400 functions, the function is configured to exit after each trigger in Pigeon. So containers and cloud resources are recycled after each serving. The function triggers are generated in a controlled rate to prevent sending all 400 triggers at the same time. The function dispatcher of Pigeon does not allow a trigger to be served if there is no CPU resources or no ready pre-warmed containers. These unserved triggers are counted as failed triggers. A total of 400 over-subscribed static pre-warmed containers are created in Pigeon before test starts.

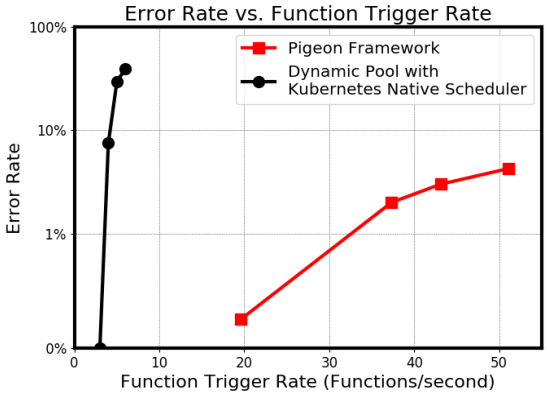

Fig. 4: FaaS Resource Efficiency Test

In Figure 4, when the trigger rate is 19 function/sec, Pigeon can serve 400 functions using limited CPU resources with only 0.25% failed triggers. When the trigger rate is at 43 functions/sec, the failed trigger is around 3%. When the trigger rate is increased to 51 functions/sec, the failed trigger is around 4.25%. For dynamic pre-warmed container pool approach used in OpenWhisk and Fission, function trigger rate varies from 3,4,5,6 functions/sec corresponding to failed trigger rate of 0%, 7.5%, 29.5%, and 39.5% respectively. From this test we can see that Pigeon can support more functions using limited cloud resources and still maintain high performance. This is an important feature for private cloud and a differentiator for Pigeon comparing to other FaaS frameworks, such as OpenWhisk, Kubeless and Fission.

C. Throughput Comparison

In this section, we measure overall throughput of Pigeon and Kubernetes native scheduling based dynamic pre-warmed container pool approach. The cluster is configured the same as section 6.1. A total of 200 empty Node.js FaaS functions are created. Tester continuously sends Restful triggers of these functions to cluster in parallel for 5 min with maximum possible speed the framework can handle. All resources in the cluster are used. The total throughput counts only successful returned triggers. For Pigeon, 600 oversubscribed multiple sized pre-warmed containers are statically created in the cluster. For Kubernetes native scheduling based dynamic pool approach, initial pre-warmed container pool size is 80.

We measure throughput for different function life span, e.g. 10, 30, and 60 sec. Once a function life cycle ends, it automatically exits. Its container is recycled. The function will be reloaded onto a new pre-warmed container when receiving a new trigger. This test mixes function first time trigger test and function subsequence trigger test. It reflects normal FaaS running condition.

In Figure 5, over-subscribed static pool approach used in Pigeon shows 3∼4 times better performance comparing to Kubernetes native scheduling based dynamic pool approach used in OpenWhisk and Fission. This is because the Kubernetes native scheduling approach uses simple one function per service mapping. When function life cycle ends and all available resources are used, subsequent triggers have to wait until existing service and container is depleted, and new Kubernetes service, Pod and container are recreated, which may take 60sec during burst trigger situation. Pigeon on the other hand, has more spared containers due to over-subscription. Pigeon function-level resource scheduler can borrow these over-subscribed pre-warmed containers when a same sized container is under recycling. Therefore, there is almost no waiting time, leading to overall high and stable throughput.

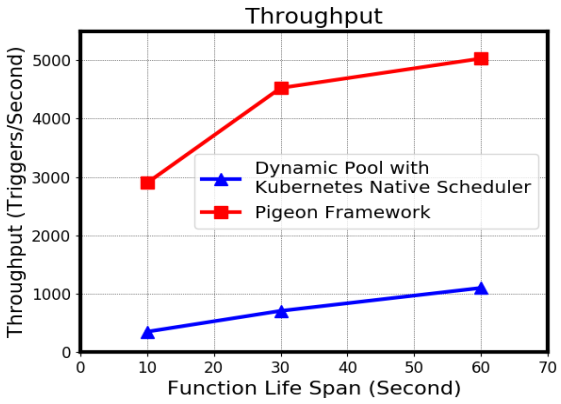

Fig. 5: FaaS Throughput Test

VII. Conclusion

In this paper, we present a new private cloud serverless and FaaS solution — Pigeon framework. Pigeon provides organization with better control, dynamics, efficiency and portability comparing to public cloud Serverless solutions. One novelty of Pigeon is the introduction of function level resource scheduler on top of Kubernetes. With function level resource scheduling, FaaS function can be directly dispatched to pre-warmed containers, which greatly reduces limitations imposed by Kubernetes and increase system performance. Overall throughput gets 3 times improvement comparing to Kubernetes native scheduler based serverless platform. Another novelty of this framework is the introduction of oversubscription-based static pre-warmed container pool. This approach eliminates the necessity of dynamically creating containers during burst function trigger situation and improve system dynamics. It also enhances function first time trigger rate by 26% to 80% comparing to AWS Lambda serverless platform.

References

[1] AWS, "Alexa Skills Kit," https://developer.amazon.com/alexa-skills-kit, 2019, accessed: 2019-06-04.

[2] Tencent, "WeChat Mini-programs Wiki," https://en.wikipedia.org/wiki/WeChat, 2019, accessed: 2019-08-21.

[3] Fission, "Open source, Kubernetes-native Serverless Framework," https://fission.io/, 2019, accessed: 2019-06-04.

[4] IBM, "Apache OpenWhisk," https://openwhisk.apache.org/, 2019, accessed: 2019-10-06.

[5] Amazon, "AWS Lambda Function," https://aws.amazon.com/lambda/, 2019, accessed: 2019-04-06.

[6] Kubeless, "Kubernetes native serverless framework," https://github.com/kubeless/kubeless, 2017, accessed: 2019-04-06.

[7] Iguazio, "Nuclio," https://github.com/nuclio/nuclio, 2017, accessed: 2019-04-06.

[8] Microsoft, "Azure Functions," https://azure.microsoft.com/en-us/services/functions/, 2019, accessed: 2019-04-06.

[9] G. Cloud, "Google Cloud Functions," https://cloud.google.com/functions/, 2019, accessed: 2019-04-06.

[10] V. Soni, *Serverless is the next evolution of a microservice architecture: New Stack Serverless Survey*, 12 2018.

[11] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, *Cloud Programming Simplified: A Berkeley View on Serverless Computing*, 2019. [Online]. Available: http://arxiv.org/abs/1902.03383

[12] M. Stein, *The Serverless Scheduling Problem and NOAH*, 9 2018.

[13] O. Alqaryoutia and N. Siyamb, *Serverless Computing and Scheduling Tasks on Cloud: A Review*, 2018.

[14] Google, "Build Actions to help users to get things done with the Google Assistant," https://developers.google.com/actions/, 2019, accessed: 2019-06-04.

[15] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, *Peeking Behind the Curtains of Serverless Platforms*, Boston, MA, 2018. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/wang-liang

[16] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, *Cold Start Influencing Factors in Function as a Service*, 10 2018.

[17] T. K. Authors, "Kubernetes," https://kubernetes.io/, 2019, accessed: 2019-04-06.

[18] OpenFaaS, "Open FaaS," https://www.openfaas.com, 2019, accessed: 2019-10-06.

[19] Google, "Google Kubernetes Engine (GKE)," https://cloud.google.com/kubernetes-engine/, 2019, accessed: 2019-10-04.

[20] AWS, "Amazon Elastic Kubernetes Service (EKS)," https://aws.amazon.com/eks/, 2019, accessed: 2019-10-04.

[21] Microsoft, "Azure Kubernetes Service (AKS)," https://docs.microsoft.com/en-us/azure/aks/, 2019, accessed: 2019-10-04.

[22] G. C. David Jackson, *An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions*, 11 2018.

[23] Dominic Kim, "OpenWhisk Scheduling Proposal," https://cwiki.apache.org/confluence/display/OPENWHISK/Autonomous+Container+Scheduling+v1, 2018, accessed: 2019-10-06.

[24] Amazon, "AWS Serverless Application Model," https://aws.amazon.com/serverless/sam, 2019, accessed: 2019-04-06.