Shared Data Plane: Tenant-Aware, Shared Context for Functions in Serverless Computing

Satwik Kolhe[1], Snehal Kamalapur[2]
[1,2]*Dept. of Computer Engineering, K. K. Wagh Institute of Engineering Education and Research, Nashik, India*

***Abstract*: Software architecture has evolved from monoliths to microservices; Serverless computing, specifically Functions as a Service (FaaS), brings in a completely new approach in terms of granularity of functional modules and their execution pattern. Tough serverless computing is understood as an event-driven model, it has a lot to offer than just deploying event-driven systems. For any application, data is an integral part that shapes the design and execution pattern. Serverless Architecture confines the size and methods in which data can be shared between individual functions. Current FaaS platforms like AWS Lambda, Azure Functions, Apache OpenWhisk; provide a limited scope in which data can be shared between individual functions; which is often addressed interchangeably as Execution Context or Shared Context. The nature of the execution context is ephemeral, which introduces additional latency to cold starts. The restricted and finite amount of space made available for execution context also confine application design, limiting the types of applications that could benefit from serverless computing. This paper addresses such issues by introducing a methodology to define tenant aware, function-specific shared data plane. This shared data plane is transparent to function code and provides localized data referencing. This paper describes the architecture of Shared Data Plane together with optimizations to Apache OpenWhisk exemplified on an inference use-case requiring a large size pre-trained model. This approach of shared data plane shows improvements to the architectural design of serverless applications with increased performance and throughput of highly parallel stateless function invocations.**

***Keywords*: Serverless Computing, Functions as a Service (FaaS), Apache OpenWhisk, Ceph.**

1. Introduction

Cloud Computing has evolved into a superset of technologies and methodologies to run applications in this internet connected world. Cloud Service Providers have largely shaped the ecosystem by constantly updating landscape of service offerings and technology to run applications, essentially reducing the application development and maintenance overheads. Infrastructure as a service model is the starting point where applications were run on shared infrastructure or cluster of virtual machines as single tier or multi-tier applications. With the introduction of managed services, all essential support functions other than business logic like, API management, Load Balancing, Database management etc. were provided as services giving rise to PaaS and SaaS offerings. Technological advancements and new software design patterns led new software architecture notably micro services and cloud-native applications. A generic trend that is evident here with respect to cloud-native application is isolation of concern and granularity of individual functional components. Rise in the number and variety of managed services is also responsible for loosely coupled application architecture. As cloud offerings mature new technologies and managed services are offered under the jargon of anything as a service. Under such circumstances, the idea of serverless computing emerged. Serverless referred to as Cloud Events [1], in many ways is the next reductive step in IaaS abstraction; replacing coder’s concerns about hardware and software dependencies with conceptually simpler function calls to act upon various other cloud services or cloud resident data sets. In broader terms, Serverless computing refers to the concept of building and running applications that do not require server management [2]. Serverless computing is majorly categorized as Backend as a Service and Functions as a Service, while the latter is usually referred to when discussed about serverless. Functions as a service is described as a fine-grained model, where application consists of one or more functions, which are deployed on a platform that are executed and scaled on demand. In FaaS at the atomic level, each application requirement is realized as an individual functions, which collectively perform as a single application as a whole.

2. Related work

Amazon Web Services, were the pioneers in providing Functions as a Service back in late 2014. AWS Lambda is an event-driven serverless computing platform that allows to run code in response to an event. AWS Lambda starts a Lambda instance supporting programming languages like Python, Node.js and Java [3] etc. AWS Lambda provides Lambda Context object that essentially allows a function to store function metadata which is limited to function properties and execution environment. Another shared object provided by AWS Lambda is Execution Context [4], which is a temporary execution environment that initializes any dependencies the function code requires. Additionally, a 512MB disk space in /tmp directory is provided to the function that can persist if this execution context is frozen. This directory can act as a cache to share data between multiple function invocations if configured.

Microsoft Azure Functions, is a similar event-driven serverless computing platform provided by Microsoft Azure.

Azure Functions [5] provide language dependent Execution context which needs to be defined within the function for the functions to be aware about it. For sharing large files Azure provides execution_context_functiondirectory (as addressed in case of python function) which cloud be shared between multiple function invocations.

Google Cloud Platform provides two services under the umbrella of serverless computing, Cloud Functions and Cloud Run. Google Cloud Functions [6] store function code and required dependencies in a read-only directory and provide a /tmp directory via tmpfs to read-write data. This /tmp directory is ephemeral and available only for individual function invocation. Google Cloud Run [7] allows subsequent use of variables by the use of global variables which are defined as part of function code. Cloud Run reuses container instances in which function is executed, but does not encourage the use of global variables as a mechanism to share data between multiple function invocations.

Other than CSPs providing serverless computing platforms under Functions as a Service model, there are opensource projects that implement serverless computing. Among these OpenFaaS, Apache OpenWhisk, Knative, Kubeless are a few. OpenFaaS implements serverless computing using kubernetes native objects, essentially starting a container environment to execute function code. Even so OpenFaaS do not provide volume mounting capabilities to implement data sharing and localized referencing for function code.

Apache OpenWhisk, is another opensource serverless computing platform which allows execution of functions – referred to as actions, in response to trigger or HTTP(s) API calls. OpenWhisk reuses container environment in which action is executed, thus global variables defined in the action code can share context information across frequent subsequent action invocations. This does not guarantee reliable context sharing and is limited only to data with small size. This approach is also not appropriate in terms of parallel execution of functions where sharing context through global variables will impact business logic.

3. Apache OpenWhisk architecture

Apache OpenWhisk is an open source, distributed Serverless platform that executes functions in response to events at any scale [8]. OpenWhisk manages the infrastructure, servers and scaling using Docker containers. OpenWhisk platform supports a programming model in which developers write functional logic (called Actions), in any supported programming language, that can be dynamically scheduled and run in response to associated events (via Triggers) from external sources (Feeds) or from HTTP requests. Apache OpenWhisk provides a platform neutral implementation of serverless computing, which can be deployed using Kubernetes, Docker Engine or as a cluster of Virtual Machine. Apache OpenWhisk, supports wide variety of programming languages, where actions are executed in a containerized environment.

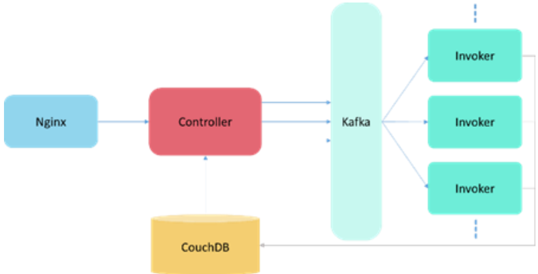

Fig. 1. Architecture of Apache OpenWhisk

Apache OpenWhisk has following key components [9],

- Nginx: A high-performance web server and reverse proxy.
- Controller: This is the main component that manages entities, handles trigger fires, route action invocation.
- CouchDB: A scalable, document-oriented NoSQL database.
- Kafka: A distributed, high-performing publish /subscribe messaging system.
- Invoker: Launching the containers to execute the actions.
- Action Containers: Actual execution of action, which is a self-contained docker container.

Apache OpenWhisk stores action properties and context information as Action Metadata. Action Metadata is transparent to action code, but not implemented with the purpose to share across function executions.

4. Shared data plane framework

A. *Shared Context*

In OpenWhisk, Shared context is a piece of information that is shared between function invocations. This information is independent of input parameters, thus being transparent to multiple function calls along the call tree. This context is made available to function code allowing inter-function communication and localized referencing. Shared context can be explicitly added to a function allowing sharing of information to functions that previously did not have.

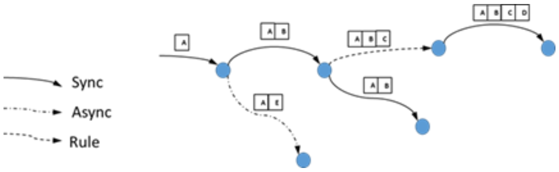

Fig. 2. Conceptual data flow of shared context

B. *Dynamic Volume Provisioning*

Kubernetes orchestrates pods, which are collection of one or more containers. These pods can be stateless or stateful. Stateless pods, as the name suggests are not persistent and data generated by these pods lives as long as the pod exists. Stateful pods on the other hand store data permanently onto disk, by using a method known as “volume mount”. In case of docker, volume mounting is a method to attach a mount point of host machine into a destination path inside the container, allowing the processes running inside this container environment to locally access this directory in the container environment, while read-writes to files are made to host machine mount-point that is persistent to host machine disk. Kubernetes generalized this concept of volume mount as persistent volumes. A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes [10]. These volumes have a lifecycle independent of the pod, essentially allowing data persistence across pod restarts. These volumes can be manually created or dynamically created by a storage class through a persistent volume claim (PVC). Persistent volume claim defines the requirement of a persistent volume which are provisioned by kubernetes via Container storage interface. Container storage interface is an abstraction provided by kubernetes to implement volume management and integration of different storage technologies like NFS, iSCSI, Ceph etc. in kubernetes. These technologies (say Ceph) is then responsible for providing objects to store data files which can be separately managed from kubernetes.

Kubernetes CSI also allow to implement storage operators that enable dynamic volume provisioning and software defined storage capabilities for provisioning persistent volumes. Rook is one example of storage operators that orchestrates Ceph abstracting the complexity of deploying ceph cluster and integrating with kubernetes CSI.

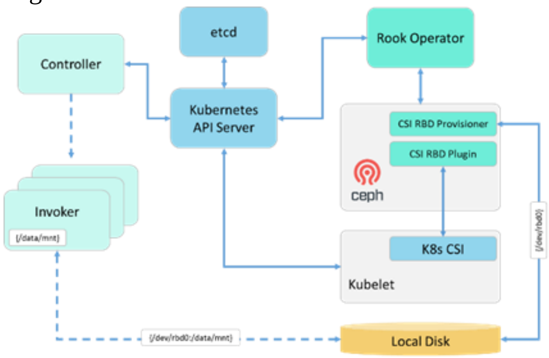

Fig. 3. Dynamic volume provisioning in kubernetes with rook

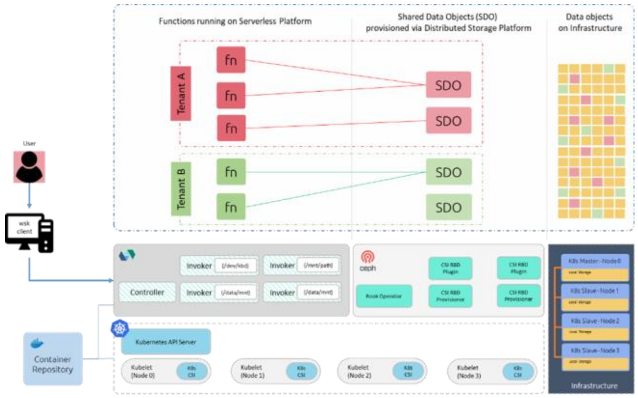

Fig. 4. Architectural approach to implement Shared Data Plane in Apache OpenWhisk

C. Proposed changes to implement Shared Data Plane

1. *OpenWhisk Controller* – Extending the invocation API (REST) to include a context field. Creation of context when an invocation request is received. Embedding the context in the activation record.
2. *OpenWhisk Invoker* - Adding the context to the environment variables of the container. Defining configuration to volume mount-points for Action Containers.
3. *OpenWhisk Client Libraries* - Extend library API with optional context parameter. Transparent copying of the client's context (environment variable) into the context field of invocation requests.

5. Experimental setup

Shared Data platform for serverless computing focuses on data objects that are shared and accessed by functions in runtime. To test the system as proposed in this paper, the major objective was to dynamically provide mount points for function containers to access data objects from those mount points. These mount points are block or filesystem volumes that store data as files in them. These volumes are mounted to function containers with read-only or read-write permissions depending upon the use-case. In this experimental setup, read-only filesystem volumes are considered for an inference use case implemented using machine learning. A major challenge that needs to be addressed is making a volume available to a function container running in a distributed environment.

Apache OpenWhisk is a collection of opensource technologies functioning as a single system in co-ordination with Controller and Invoker. Apache OpenWhisk can be deployed as a cluster of virtual machines where individual services like Kafka, CouchDB and Nginx can be configured in a highly-available cluster. Apache OpenWhisk deployment using docker containers deployed by Docker Compose, allows to setup a development environment with default settings. Another deployment method is using a Container Orchestration Platform like Kubernetes [11], which allows greater configuration options and allows deployment of production ready environment.

Table 1
Cluster configuration of Experimental Setup

Hostname	K8s version	OS Image	Kernel Version	Docker version	Ceph version
bm-k8s-master	v1.16.4	CentOS 7	3.10.0	v19.0.3	14.2.3
bm-k8s-slave-1	v1.16.4	CentOS 7	3.10.0	v19.0.3	14.2.3
bm-k8s-slave-2	v1.16.4	CentOS 7	3.10.0	v19.0.3	14.2.3
bm-k8s-slave-3	v1.16.4	CentOS 7	3.10.0	v19.0.3	14.2.3

To perform tests as proposed in this paper, Apache OpenWhisk was deployed using kubernetes and helm charts [12] on a cluster of four bare-metal nodes. Updates enabling shared data object management capabilities were pushed to controller and invoker modules as custom docker containers. For dynamic volume provisioning a storage-class was implemented that abstracts volume provisioning as a set of APIs interacting with Ceph – distributed storage platform.

Hostname K8s version OS ImageKernel Version Docker versionCeph version

In this experimental setup, Apache OpenWhisk version 8eb922f and Rook v1.2 was used.

The user while creating a function also need to create a shared data object. This shared data object is created using modified ‘wsk’ cli, which is also used to create a function. The controller creates a function and creates associated shared data object. Metadata of this shared data object is stored as part of function annotation in CouchDB. The controller is responsible to create a volume on the distributed storage platform and initialize it with the provided data files. The appropriate mount points are updated in function metadata after successful provisioning of this volume in CouchDB.

The function can be invoked using wsk cli or by making a HTTP request using function’s API. This function API can be defined using the same wsk cli. Regardless of the method used to invoke the function, the controller instructs invoker instance to execute the request. The controller is also responsible to instruct the container orchestration platform to mount the shared volume when invoker instance is started. Depending upon the programming language the invoker instance is chosen to execute the function request. The invoker decodes function metadata provided by Controller and extracts information like default parameters, function resource limits, execution timeout and shared data object mount point. As the shared data object volume is locally mounted inside invoker’s container environment, the function makes local references to those files.

For the inference use case discussed above, a function is implemented in python which detects objects in an image. This function is written in python, which uses tensorflow r2.0, numpy, matplotlib and pillow python libraries. The function accepts an image as input and output a json object that contains percent probability of objects that are present in the input image. This function is created as a docker action on the serverless platform discussed above. The function requires a large 1.5GB ‘model.h5’ file, which contains weights required for object detection task. A shared data object is created which is initialized by this model.h5 file. A POST API is created for the inference function which is used to invoke it via API call. The size of docker action which included function code and required libraries was around 1.3GB.

Multiple function invocation were called to test cold-start performance of the function. These function calls were made using curl and bash script. In the interval of 15 minutes, 5 consecutive function API calls were made, capturing the request duration – start and end time of the HTTP request. The request duration included the time required for the function to process the image and output the results.

To generate baseline results, another test was performed on a standard Apache OpenWhisk platform, which did not support shared data objects. Due the same reason the docker action created for the inference function, had a size of 2.8GB where 1.5GB is the size of the model.h5 weights used by the function and remaining 1.3GB was the size of tensorflow and other libraries included as code in docker action. On similar grounds, multiple function invocation were called to test the cold-start performance of this standard docker action. The same curl and bash script was executed in intervals of 15 minutes to make 5 consecutive function API calls.

6. Results and discussion

Apache OpenWhisk allows to create functions that require external libraries that are not available as standard libraries for any specific programming language. In the case of inference function discussed in experimental setup, these libraries included tensorflow, numpy etc. To deploy such function OpenWhisk allows two options, deploying function code and required libraries as a zip package or deploying the function as a docker container image, the later is suitable in case of much more complex function implementations as in the case discusses here. To deploy a function as docker container, called as docker action, the container needs to be built using a Dockerfile, which essentially packages all files as per the instructions defined in that Dockerfile and generates a docker image. For the inference function, tensorflow and other libraries were installed while building the docker action container image. The size of these file in itself is around 1.3GB. Additionally, the inference function required weights file (model.h5) which generated by training the machine learning model. Thus for the function to execute this weights file need to be packaged inside the docker action container, increasing the image size of the action container to a total of 2.8GB. This increase in size of docker action container image becomes a major issue considering the fact that every time the inference function is executed this docker image needs to be pulled from a remote docker container repository leading to unnecessary network utilization.

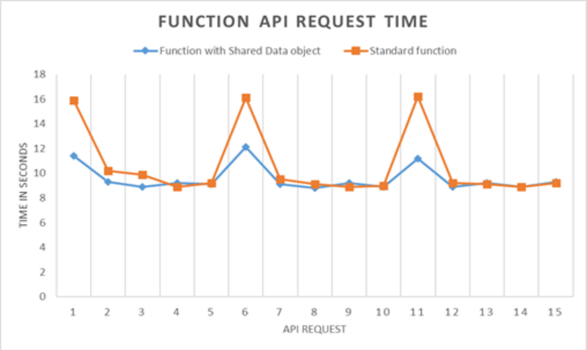

Fig. 5. Graph showing function request time for 5 consecutive API calls made in the interval of 15 minutes

Thus the shared data object allowed the function to be deployed as two separate entities of code and data, where this data is essentially only required to initialize the function variables. Due to which the size of function’s docker action container was reduced by almost 46% reducing the data

Table 2
Experimental results showing HTTP request duration, actual function execution time and cold-start latency

Environment	API Call	Request Duration	Fn execution time	cold-start latency
Serverless Platform with Shared Data Plane Framework	1	11.4	9.9	1.5
	2	9.3	9.1	0.2
	3	8.9	8.7	0.2
	4	9.2	9	0.2
	5	9.1	8.9	0.2
	6	12.1	10	2.1
	7	9.1	9	0.1
	8	8.8	8.7	0.1
	9	9.2	9	0.2
	10	8.9	8.7	0.2
	11	11.2	9.9	1.3
	12	8.9	8.7	0.2
	13	9.2	9.1	0.1
	14	8.9	8.7	0.2
	15	9.3	9.1	0.2
Standard Serverless Platform	1	15.9	9.8	6.1
	2	10.2	9.9	0.3
	3	9.9	9.7	0.2
	4	8.9	8.7	0.2
	5	9.2	9	0.2
	6	16.1	9.9	6.2
	7	9.5	9.3	0.2
	8	9.1	8.9	0.2
	9	8.9	8.6	0.3
	10	9	8.8	0.2
	11	16.2	9.6	6.6
	12	9.2	8.9	0.3
	13	9.1	8.8	0.3
	14	8.9	8.7	0.2
	15	9.2	9	0.2

footprint of the function created in serverless platform. This is also ideal for frequent updating of function code, where changes are only made to the code and the data stays the same.

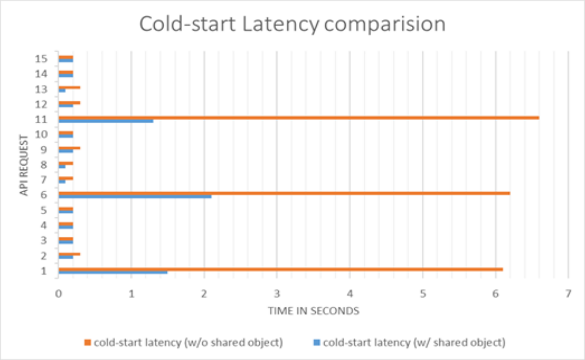

Fig. 6. Graph showing comparison between cold-start latency

In serverless computing, cold-start is referred to as the time required to initialize the function environment where that function will be executed. In the experimental setup discussed above, network bandwidth was of 1GBps, which is a very ideal case from the point of view of infrastructure. In other cases, where network bandwidth is less than this would hamper the cold-start latency while executing a function.

The table above show results from experiment conducted in two different environments. First tests was conducted on serverless platform with shared data plane framework and the second test was conducted on standard Apache OpenWhisk serverless platform.

Due to reduced data footprint, this cold-start latency is significantly decreased. From the experimental results, for a function created with a shared object, this cold-start latency is decreased to by 67% as compared with the latency observed when the same function created as a single docker action on standard Apache OpenWhisk environment.

Table 2 Experimental results showing HTTP request duration, actual function execution time and cold-start latency

This capability to separate code from shareable objects in serverless computing creates an opportunity for application to run on this new programming model which previously was not possible. As demonstrated here, serverless computing allows to implement a highly scalable object detection function capable of handling parallel requests. Data isolation implemented at the level of Apache OpenWhisk, Ceph and Kubernetes reduces the security risks associated in any computing platform.

The results discussed here are showcasing the benefits of shared data plan in serverless platform from only one single use-case of a machine learning workload. Where as in other cases this could lead to highly granular implementation of application which could benefit from separating code and data objects. Nonetheless this requires adoption of new serverless programming model which has a learning curve associated with it. This makes it difficult for developers to adopt serverless computing as a mainstream application development strategy even after experiencing its benefits. Serverless computing also does not fit in right for every application because of latency associated with executing a function. On the brighter side serverless computing provides zero server maintenance, out of the box scalability and cost savings.

7. Conclusion and future work

This paper introduced the concept of Shared Data Plane, a framework allowing to create shared objects that enable separation of code and data objects required by a function defined in serverless platform. The work described in this paper also addressed the problem of cold-start associated with serverless platform, and demonstrated improvements in performance of serverless platform to server functions in response to API calls. Due to separation of code and data, the data footprint of functions deployed on Apache OpenWhisk was reduced improving developer experience and management of functions. Serverless Computing is a new programming model which is largely considered to be event-driven restricting the types of applications that can be developed using serverless architecture. The concept of Shared Data Plane brings in opportunities with respect to the type of applications that can be deployed using serverless technology. As exemplified in this paper, a machine learning use-case can be implemented on serverless platform, which previously was hindered due to constraints with function deployment methods and function execution efficiency bottlenecks affecting the end user experience.

The experiment described in this paper, shows reduction in docker action image size, leading to reduced cold-start latency. But evidently in this case, the data footprint of inference function which is around 1.3GB, is still considerably large because of the fact that required libraries take that much space. With the goal of reducing this data footprint, shared data plane could also provide sharing these libraries across multiple function execution environments further reducing the deployable function size. To achieve this serverless platform can utilize Docker’s capability to mount multiple volumes to the same container. But these libraries need to be available to the function with minimum nanosecond latency as possible, as the function logic could require access these libraries frequently, and any latency introduced in this would lead to decrease in function execution efficiency.

Acknowledgment

The authors would like to thank all collaborators and supporters of this work. Satwik K. would like to thank K. K.W.I.E.E.R to provide an opportunity and their collaboration. Satwik K. would also like to thank ESDS Software Solution Pvt. Ltd. to sponsor this research work.

References

[1] Garrett McGrath in Cloud Event Programming Paradigms, 2016 IEEE 9th International Conference on Cloud Computing.
[2] CNCF Serverless Whitepaper https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf
[3] AWS Lambda FAQs, What programming language does AWS Lambda support? https://aws.amazon.com/lambda/faqs/.
[4] AWS Lambda Execution Context https://docs.aws.amazon.com/lambda/latest/dg/running-lambda-code.html.
[5] Retrieving information about the currently running function https://github.com/Azure/azure-functions-host/wiki/Retrieving-information-about-the-currently-running-function.
[6] Cloud Function Execution Environment, File system, as accessed in January 2020, https://cloud.google.com/functions/docs/concepts/exec.
[7] Development Tips, Using Global Variables, https://cloud.google.com/run/docs/tips.
[8] What is Apache OpenWhisk? https://openwhisk.apache.org/.
[9] Serverless and OpenWhisk Architecture, https://www.oreilly.com/library/view/learning-apache-openwhisk/9781492046158/ch01.html.
[10] Persistent Volumes, in Kubernetes https://kubernetes.io/docs/concepts/storage/persistent-volumes/.
[11] OpenWhisk deployment on Kubernetes. https://github.com/apache/openwhisk-deploy-kube.
[12] Helm Package manager Quickstart Guide. https://helm.sh/docs/intro/quickstart/.