National College of Ireland

Using Redis for persistent storage in serverless architecture to maintain state management

MSc Research Project
Cloud Computing

Ankit Kumar
Student ID: 20149158

School of Computing
National College of Ireland

Supervisor: Adriana Chis

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Ankit Kumar
Student ID:	20149158
Programme:	Cloud Computing
Year:	2021
Module:	MSc Research Project
Supervisor:	Adriana Chis
Submission Due Date:	31/01/2022
Project Title:	Using Redis for persistent storage in serverless architecture to maintain state management
Word Count:	XXX
Page Count:	18

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	31st January 2022

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Using Redis for persistent storage in serverless architecture to maintain state management

Ankit Kumar
20149158

Abstract

Serverless computing has been the talk of the decade, it provides a small runtime container that executes a function or task without the need of resource management, it is a type of Platform as a Service (PaaS) but at a much smaller functional level. The first cloud serverless service was introduced by Amazon Web Services (AWS) in 2014, earlier it used to have a limit of 25 concurrent function limitations but now it can provide thousands of functions at a time because of the demand in the technology. Even with the dynamic growth of serverless computing, there have been some limitations in architecture that are still a major concern. Limitations such as inter-communication between functions, lack of persistent storage, resource selection for certain tasks e.g. allocating GPU for machine learning tasks, etc. In this paper, we will be discussing how we can implement a serverless Redis container to be accessed by serverless functions when it is needed. The approach will spawn a Redis container along with OpenWhisk actions and when the function is completed, the Redis container will be shut off to release the resources, similarly to how the serverless function works. The experiment is performed in an OpenStack instance that has similar machines as Amazon Web Services (AWS). We compared the connection of the Redis container with OpenWhisk actions against AWS Lambda and ElastiCache Redis. We measured the time of functions execution and the operations are done on the Redis database. From the experiments conducted, we concluded that our approach performance surpasses Lambda and ElastiCache stack by 85%, in production scenario the margin will be less though for the sake of the experiment we maintained the best configuration possible for the OpenWhisk framework and Redis container.

***Keywords*—** OpenWhisk, Serverless, Amazon Web Services (AWS), Redis, ElastiCache

1 Introduction

Serverless computing is adopted widely since the last decade, it is a commercial service provided by cloud providers to execute event-driven functions that require small resource allocation. Along with serverless, containers and micro Virtual Machines (microVMs) have also gained a lot of popularity. Companies need fine-grain control and security over their applications. Most companies are shifting from a monolithic architecture to microservices, which makes their application much durable and fault-tolerant. Containers can start in a second and destroy quickly because of their lightweight kernels. The application hosted in the containers can be preserved in the image and distributed among multiple instances. Along with several benefits of serverless functions, the most popular will be its scalability. A traditional VM-based architecture will take a lot of effort and time to set up the instance with networking and storage configurations, in serverless we do not have to worry about the resources and configurations. Figure 1 shows the difference between traditional VMs and containers.

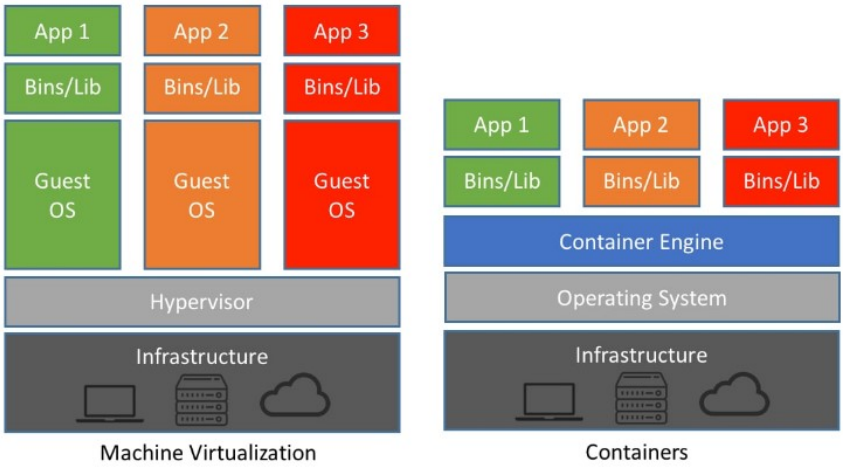

Figure 1: VM vs Containers. Source[1]

Cloud services mostly use a pay-as-you-go pricing model, in which the customer is charged on the uptime of resources, not on the usage of those resources, so there may be a chance that a company is paying more than it should according to their requirements. Serverless computing may provide a cost-effective approach as it billed on the execution time of containers and the number of requests. Its charging metric is counted from 0.1 seconds, though many commercial cloud services still use the hour for metrics. Google and Microsoft were using per-minute billing for their virtual machine instances but recently Amazon Web Services also introduced per-second billing[2], but still, these instances will cost every second even if there is no program running on them.

Serverless term is misleading as it is run on a physical machine but the user does not have to do resource management and infrastructure configurations. Function-as-a-Service (FaaS) and event-driven computing come under Serverless computing, which offers event-driven architecture for micro-services in which an event generated by other services or resources triggers the functions/compute. The event can be any service that supports an API to notify other resources, events such as deleting or adding data in a database, updating a file in block storage, a notification from the Internet of Things (IoT), etc. Some events are generated at certain intervals that are predictable, such as health checks of a system, polling on a resource, updates on notification, etc.

[2]AWS EC2 Per-second billing: `https://aws.amazon.com/blogs/aws/new-per-second-billing-for-ec2-instances-and-ebs-volumes`

Every serverless function is isolated in a container with its dedicated resource pool. Due to the container lightweight VM, we can instantiate thousands of functions in seconds and achieve much higher parallelism as compared to traditional VM. Since the last decade, mobile and IoT have gained traction because of smart homes and smart city concepts. To handle such an amount of events, we would require a solution that can scale accordingly. (Baldini et al. 2016) observed that using a single instance in multiple events workload can degrade the performance of the application.

1.1 Limitations of FaaS

Regardless of the several benefits of Serverless computing, one should not use it in every application. The application use-case should be considered, some examples of applications that should not use serverless are applications that listen to events like HTTP all the time, parallelly dependent functions that require passing to each on a shared network, machine learning, memory-intensive applications, etc. Currently, Serverless does not support persistent data storage, though we can use external storage such as AWS Simple Storage Service (S3), ElastiCache, AWS Relational Database Service, etc. We do not have control over the hardware architecture of the compute, if we need to run applications that are hardware agnostic e.g. running applications on limited number of CPU or RAM, serverless would not be able to satisfy the need.

1.2 Need for persistence

As mentioned above, Serverless can not host a database on its own, it does have ephemeral storage and global variable storage. A global variable can be shared with other functions but they all have to be on the same session. For instance, in AWS Lambda [3] if we invoke a function with global variable e.g. `global.counter=0`, the function will be bootstrapped in a container and that container may or may not be used again for the same function to increase performance and efficiency. If we get the same container we can access the global variable, so the functions running on or around the same request will have global variable access.

In this paper, we will be introducing and implementing Redis Cache Database as a serverless instance. The instance will be instantiated along with the cold start of functions. The goal is to provide persistent storage for serverless functions with the flexibility and cost-efficiency of serverless architecture. We will be using Apache OpenWhisk [4] as our FaaS platform, which uses Docker for containerization and Kubernetes for container orchestration. We will be monitoring the execution time of the functions as they connect with the Redis instance and compare the same function with AWS Lambda and ElastiCache Redis [5] time.

2 Related Work

This section aims at the evaluation and study of recently published state-of-the-art papers. All these papers are closely related to the work we are conducting, thoroughly evaluating these studies will provide

[3] AWS Lambda: `https://aws.amazon.com/lambda`
[4] Apache OpenWhisk: `https://openwhisk.apache.org`
[5] AWS ElastiCache Redis: `https://aws.amazon.com/elasticache/redis`

us with valuable knowledge for better decision making and help us to streamline our process in methodology and evaluation. We will start subsection 2.1, which is an evaluation of serverless environments (FAAS).

2.1 Evaluation of Serverless Environments

Serverless is an execution of small functions, already configured in small resources and images. The containers on which the functions are run are lightweight and fast, which means it can be instantiated and destroyed in seconds whereas the application is stored in a distributed environment. Lee et al. (2018) has evaluated various FAAS providers in terms of performance and cost. The performance is measured in concurrent throughput, CPU-intensive workloads, disk-intensive workloads, network-intensive workloads, elasticity, and ease of deployment. The experiments are carried on AWS Lambda, IBM OpenWhisk, Azure Functions, and Google Cloud Functions. Overall, all of them exceed in at least one aspect of performance than their competition. According to the paper, AWS Lambda has better Floating Point Operations Per Second (FLOPS) than the rest, IBM OpenWhisk performs better in I/O operations, where Microsoft Functions couldn't even manage to complete the task in time and failed. A similar report was also published for the evaluation of FAAS in different service providers [20], but the review in the report was not detailed enough and did not take into account the workloads of CPU, Network, I/O, etc. Since the report was published there have been many improvements in this area such as the limit of memory allocation on Lambda has been increased to 3GB and more runtime languages have been added. As per the report AWS Lambda has more configurations, is better performing, and has more language support, so we will be using it in our experiments, supported languages for different serverless platform is shown in Figure 2.

	AWS Lambda	Azure Functions	GCP Functions	Apache OpenWhisk
JavaScript(node.js)	Yes	Yes	Yes	Yes
Java	Yes	Yes	No	Yes (Partial)
C#	Yes	Yes	No	No
Python	Yes	Experimental	No	Yes
PHP	No	Experimental	No	Yes
Go	Yes (Partial)	No	No	No
F#	No	Yes	No	No
Swift	No	No	No	Yes

Figure 2: Serverless programming languages support

2.2 OpenWhisk startup evaluation

FAAS functions are meant to be small which does not require a dedicated listener on our application architecture. The smaller the function and its dependencies, the faster it will boot from the cold-start and complete the designated task. Likewise, if the function has dependencies on heavy libraries, the cold start will be much slower Abad et al. (2018). The report Quevedo et al. (2019), evaluates how much difference will it make if we use heavy functions on a cold start using OpenWhisk as a FAAS platform.

Java [6] and JavaScript are used for image resizing that will test the function to its limits in cold start. There will be two scenarios: cold start and hot start in two different languages with default settings and optimized configuration of OpenWhisk. The paper shows that with optimal settings OpenWhisk can perform 38% better than the default configuration. We are using OpenWhisk as a FAAS platform due to its granularity in configurations and it is a tried and tested open-source platform to run functions.

2.3 Ephemeral storage in Serverless

Serverless architecture promotes ephemeral storage because the child processes and the task's local file system are limited to the lifetime of the task. In this paper Klimovic et al. (n.d.), the author proposed three different approaches to tackle the limitation of the serverless stateless nature. They introduced three storage options in three different analytics operations and compared them in terms of throughput, I/O size, and data access frequency. AWS Lambda is used as the serverless platform, three storage options used in the report are - AWS S3, AWS ElastiCache Redis, and Apache Crail with ReFlex (Flash storage distributed system). ReFlex achieves low latency and high throughput using data panel kernel, it provides local flash storage performance in remote environments Klimovic et al. (2017). MapReduce, Parallel software build, and video analytics operations were used in Lambda to gather the metrics from different storage options. According to the results, S3 is best when there is less throughput and big files. Using ElastiCache is a bit costly but can handle large throughput with much latency, read/write speed of Redis is much lower than that of S3, e.g. Redis – 230µs/232µs and S3 12.1ms/25/8ms. But if we implement a flash storage system, it can provide a balance in both speed and cost, along with an option of Metadata lookup that can be used in several applications, the only downside of the flash storage would be asymmetric read/write.

2.4 Multiple storage options with resource management

Traditional architectures require long-running storage nodes/clients that enable intermediate data transactions over a network Zaharia et al. (2012). However, in the serverless approach, there are no long-running applications, and local storage is not managed by the architecture. This makes it difficult to communicate directly between different serverless tasks/functions. Therefore, we have to opt for traditional storage such as S3, databases (MySQL, MariaDB, etc.), and distributed cache (Redis, Memcache). Filesystems and No-SQL storage provide scalable and long-term storage options that do not prioritize performance and cost. In paper Klimovic et al. (2018), they introduced Pocket, which enables resource management in data storage with multiple options, such as HDD, DRAM, and NVMe flash. There have been studies Yadav et al. (2018) that shows resource management in the different storage device is possible via carefully monitoring the usage of the application and adapting the most appropriate solution amongst various storage options. The pocket takes advantage of this approach and provides a solution that is scalable by default, has better performance, and is cheaper than the traditional approach of long-term storage. It has three different components responsible for optimal performance: metadata, control, and data planes. Metadata tracks/monitors tracks in the data planes across different nodes. Data planes contain data storage. And, control planes manage resources, data sizing and, clustering. According to the report Klimovic et al. (2018), it provides cheaper cost and performance than S3 and traditional storage by 40x and 10x respectively. Furthermore, Pocket can be implemented in various EC2 instances by default. The downside of Pocket is that it is not a proper serverless approach, it does require storing data in various nodes and storages (HDD, DRAM, or NVM flash).

[6] Java: `https://www.java.com/en`

2.5 Shared network file system

The current file operations in FAAS should be handled within the time limit of the function execution. It supports object-level storage which is not optimal for heavy data storage, to better support FAAS, block-level storage should be implemented that enables byte-level manipulation within the function. The paper [16], adoption of Network File System (NFS) can improve data sharing between different tasks on serverless functions. NFS can be pre-mounted and prepared without any dependencies of Software Development Kits (SDK). The author of the paper has performed experiments on AWS Lambda and Elastic File System (EFS) with NFS. Because we are using block-file systems the data can be accessed in bytes therefore the performance and control can exceed the traditional storage options. The bandwidth of the block storage should be pre-assigned in the configurations while initiating EFS. It should be configured carefully as it can bottleneck the storage if there are too many functions trying to access the storage at the same time. In terms of Read/Write of data, block storage surpasses most of the storage options but due to its limitation in scalability, it may or not be a good choice for many applications.

2.6 Docker vs Firecracker

Docker and AWS Firecracker are the underlying container technologies used in OpenWhisk and AWS Lambda respectively. OpenWhisk uses Docker [7] along with Kubernetes [8] to orchestrate containers and spawn new containers as per the request for new functions to execute. Docker is the industry-leading containerization technology that enables developers to deploy secure and manageable applications without worrying about scalability and vendor lock-in.

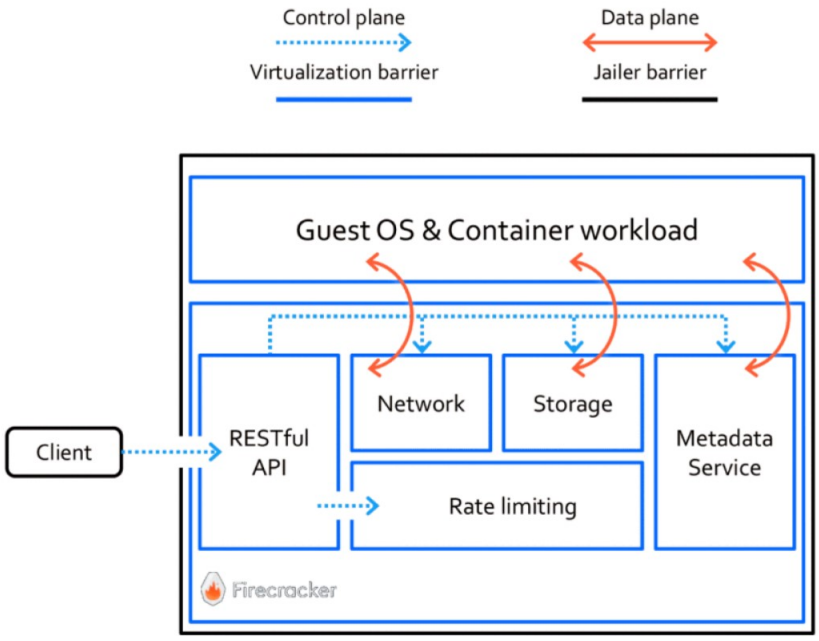

Figure 3: Architecture of AWS Firecracker Mocanu et al. (2021)

Firecracker [9] is also a containerization technology built from the ground up by AWS with security

[7]Docker: `https://www.docker.com`
[8]Kubernetes: `https://kubernetes.io`
[9]AWS Firecracker: `https://firecracker-microvm.github.io`

as a priority. It serves MicroVMs on top KVM-based virtualization that provides more security than traditional virtual machines, as shown in Figure 3. As it was developed from scratch, the team included only the necessary components and removed all non-essential functionality that reduces the attack surface area. Firecracker is used in both AWS Lambda and AWS Fargate. Both are open-source, built by giants of cloud computing, adopted by thousands of developers and companies. It secure by code as it implements a restrcitive mechanism called Jailer, which disables guest user to inject code in the Virtual Machine Monitor (VMM). It strips the guest profile privileges and sets a restricitive *seccomp-bpf* profile Agache et al. (2020). The main difference between the two is the ease of deployment for small companies. Firecracker is more complex to deploy as compared to Docker, Docker provides good documentation and a really helpful CLI, whereas to deploy Firecracker we need to have good knowledge about kernels and images. There is a trade-off, as Firecracker has its MicroVM that is lightweight and built for the sole purpose of secure and fast containerization, it is much faster than Docker (even alpine images). For our report, we will be using OpenWhisk that uses Docker, so that we could implement the containers as per our needs.

2.7 Docker Virtual Machine evaluation

The paper Preeth et al. (2016), has concluded tests on Docker and HostOS (Ubuntu 12.04) with various benchmarking tools, such as Bonnie++ [10], psutil [11]. The author is trying to find out the performance difference between Docker VMs and traditional VMs. Bonnie++ evaluates disk I/O performance, it performs 40 million invocations on the disk and provides read and write average speed. In the given test Docker VM was slightly slower than HostOS as HostOS has more resources to spare than Docker. The Docker output and input rates were 507 Kbp/s and 1351 Kbp/s respectively, whereas HostOS input and output rate was 536 Kbp/s and 4388 Kbp/s. Psutil tool is used to evaluate memory utilisations, CPU times, disk usage, and network I/O counter. In these tests, Docker and HostOS both have almost similar results. Due to Docker's caching mechanism, it was able to provide a better cache than HostOS. Even if Docker does not provide similar performance as HostOS, it does provide environment isolation, security Ahamed et al. (2021), and flexibility for deployment.

2.8 Latency in serverless functions and remote databases

For persistence, Serverless functions need to have external data storage such as DynamoDB, MySQL, MariaDB, etc. Serverless functions are mostly preferred if the application needs to execute tasks that are small and quick to complete. If we add more complexity by adding a database, it will have latency issues as the database can not be in the same container as serverless functions. AWS provides various storage options and all of them are hosted remotely. In paper (Ghosh et al. 2020), the authors have compared latency in serverless with external database and EC2 stack. They have tested this experiment in five different AWS regions, namely Mumbai, California, London, Singapore, and Canada central. They used two approaches, a simple database CRUD application build on python's Flask and a complex application with critical data analytics pipelines Bhattacharjee et al. (2019). The results concluded that the EC2 stack has 14 times less latency than that of AWS Lambda and DynamoDB stack.

To improve latency, they have added in-memory caching e.g. ElastiCache Redis Figure 4. But it also has an overhead latency in the connection.

[10]Bonnie++: `http://www.coker.com.au/bonnie++`
[11]psutil: `https://pypi.python.org/pypi/psutil`

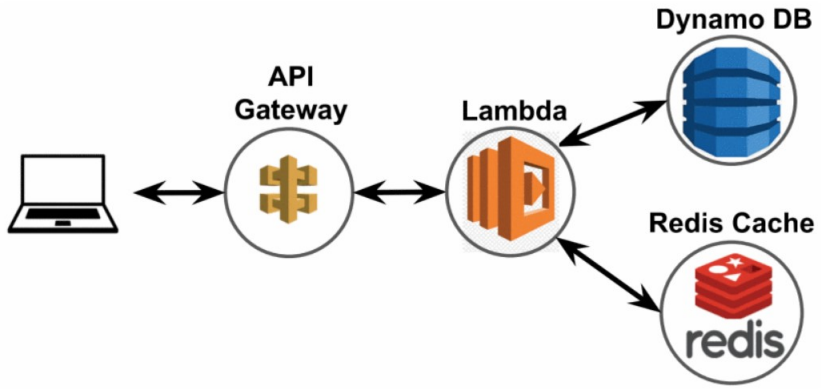

Figure 4: Redis cache to improve latency (Ghosh et al. 2020)

3 Methodology

In this section, we will be going to discuss the different processes and techniques to carry forward our experiments. Our experiments are divided into two stacks: Apache OpenWhisk + Redis container and AWS Lambda + ElastiCache Redis.

3.1 Apache OpenWhisk and Redis

We will start by setting up the cloud server instance, for the experiments we chose `m1.xlarge`instance in OpenStack [12], which is an older generation in AWS EC2 instance types[13], we will discuss more about OpenWhisk in 4.1. The instance boasts 8 vCPUs and 16GB RAM which is required for running bulk serverless functions and Redis altogether. Once the server is initiated we will be setting up the Apache Openwhisk framework to run our functions. We will be using OpenWhisk Command Line Interface (CLI)[14] to manage the global variables required to run the framework.

We will be creating two artifacts in NodeJS[15]: using OpenWhisk client library to create, update and invoke serverless functions along with Docker client library to control the lifecycle of docker container (Redis) and another artifact would be our serverless function where we will be connecting the Redis container, injecting data in the Redis, reading them, and deleting them before the function ends. We will be recording the time taken for all the queries, connection time, complete duration of the function, and spawning of the Redis container. The function will be accepting the Uniform Resource Locator (URL) and port number as parameters.

There will be a User Interface (UI) for the input of various parameters such as OpenWhisk API host, API key, name of the function e.g. used as an identifier in OpenWhisk, the path of the zip file of the function, type of runtime e.g. `nodejs:12`, and the total number of invocations. We will not be using OpenWhisk activation polling to get the active results of the functions in real-time, but we will be fetching the data from the logs that are generated after the full activation of the function. The UI will give us the result of each time frame recorded that is mentioned above concerning the activation

[12]OpenStack: `https://www.openstack.org`
[13]Older genration AWS instances: `https://aws.amazon.com/ec2/previous-generation`
[14]Apache OpenWhisk CLI: `https://github.com/apache/openwhisk-cli`
[15]NodeJS: `https://nodejs.org/en`

identifier.

3.2 AWS Lambda and ElastiCache

For practical comparison, we will be using AWS Lambda and ElastCache to run the same function which is being used in the previously mentioned stack, with a slight change in parameters as we would be needing to connect the ElastiCache Redis instance. We would also have to update the code according to the Lambda documentation[16] there is a specific naming convention that we have to follow to run a function in Lambda. According to the documentation, the execution file should be in the root, the name of the file should resemble the invoke handler configuration e.g. default is `index.handler`, therefore filename should be `index.js`, and we will have to add a handler method in the global exports module e.g. exports.handler. After all the required changes we will deploy the code in the Lambda and should be able to connect to the AWS Redis instance. The choice of Redis instance is `m6gd.xlarge`, which has 4 vCPUs, 16 GB RAM, and network bandwidth up to 10 Gbps Table 1. Both Lambda and Redis share the same Virtual Private Cloud (VPC). Lastly, we test the function we will get the same time durations objects which are to be expected. A similar stack has also been proposed in 2.8, but they were using Redis for caching data from DynamodDB to achieve performance in high throughput scenarios.

Table 1: Comparision between m1.xlarge and m6gd.xlarge instances

Name	**vCPU**	**RAM**	**Network Bandwidth**
m1.xlarge	8	16	upto 6 Gbps
m6gd.xlarge	4	16	upto 10 Gbps

We will be invoking the Lambda functions from the Simple Notification Service (SNS)[17], the number of invocations should be the same as the above stack so that we could have a fair comparison. Lambda function will return the time of the queries but we will have to use AWS CloudWatch to fetch the duration of the complete function cycle.

Once we get the outputs from both the stacks we will be comparing them both with time and cost metrics. We will calculate the average and standard deviation of all the functions invocation and queries time.

4 Design Specification

The goal of this paper is to have persistent data sharing between serverless functions. Serverless functions can only keep the data in their scope or session of functions. If there is a need to share the data we will have to use external data stores as explained in 1.1. Our design consists of a container that will have a database e.g. Redis, that will spawn along with our serverless functions and shuts off as soon as the functions are finished. To invoke serverless functions in the cloud environment we will be using Apache OpenWhisk and Docker container for Redis, these both technologies will be controlled by a custom API built on NodeJS. To have a benchmark with the results acquired by the above-mentioned technologies

[16]AWS Lambda NodeJS: `https://docs.aws.amazon.com/lambda/latest/dg/nodejs-handler.html`

[17]AWS SNS: `https://aws.amazon.com/sns`

we will be conducting the same experiments with production-ready AWS cloud services such as AWS Lambda and AWS ElastiCache Redis.

4.1 Apache OpenWhisk

Apache OpenWhisk is an open-source platform that runs serverless functions from events at any scale. It manages the server and infrastructure using Docker, which only requires an initial setup. It can listen to feed on popular sources and trigger an action which is a logical function written by a developer in any supported language. The source can be any service that supports subscription or webhook model, some of the services are Kafka, RSS feeds, Cloudant, push notifications from the web or mobile applications. The supported languages are Go, NodeJS (JavaScript), Python, Java, PHP, Ruby, Swift, and .NET Core. Since it builds its components in containers it can be easily deployed in various platforms such as Compose, OpenShift, and Kubernetes. The action can be invoked via triggers, CLI, or HTTP requests. CLI and OpenWhisk client uses HTTP requests in the background. The model is divided into three components, namely triggers, rules, and action.

4.1.1 Action

Actions are programming language agnostic which means action can be created, invoked, and managed regardless of the language it is written in. Just like any other serverless function, actions are also stateless which can only be invoked through defined means such as trigger or HTTP requests.

4.1.2 Trigger

Triggers are the predefined channels that are subscribed to an external source or feed.

4.1.3 Rule

Rules have one to one relationship with triggers and actions, they act as a mediator and they can be configured to filter out unwanted events that may cause over invocations of actions. Every time a trigger catches an event the rule will be responsible to fire the actions. Figure 5 depicts the OpenWhisk programming model.

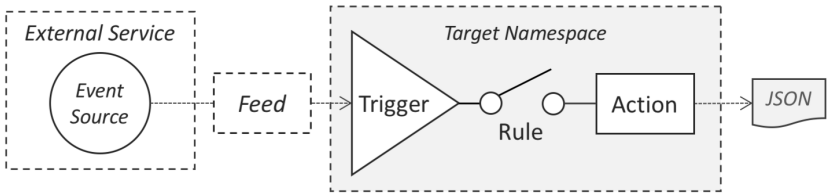

Figure 5: OpenWhisk programming model. Source[18]

4.2 Redis

Redis is an open-source in-memory data storage system mainly used for caching. It supports data structures such as hashes, lists, strings, sets, bitmaps, geospatial indexes, hyperlogs, and streams. It provides built-in on-disk persistence, Least Frequent Used (LFU) and Least Recently Used (LRU) eviction strategies, replication, transactions, and high availability via Redis Sentinel [19]. It is written in ANSI C which means it will work in most POSIX systems such as OS X, Linux, BSDs, etc without any dependencies. It is an in-memory storage system which makes it faster than conventional disk-based storage, but as it is stored in the main memory and the main memory is flushed out at system boot therefore it requires permanent storage. There are multiple persistence options provided by Redis.

4.2.1 Append Only File (AOF)

AOF writes every request to the log in real-time. The database is reconstructed using the logs at server startup, hence the original data is never lost. It can be run in the background if the persisted data becomes too big.

4.2.2 Redis Database (RDB)

In RDB we can specify time intervals to create snapshots of the database and save them on the disk. This option is mostly preferred as we have the flexibility to write the data to external sources such as AWS Simple Storage Service (S3).

4.2.3 No persistence

If we want we can disable the persistence and just keep the data until the server is running, the data will be lost after the server shutdown.

We will be using RDB persistence to store the data on the disk each time before the Redis container is destroyed.

4.3 Docker

As mentioned above 4.1, OpenWhisk uses Docker to manage the infrastructure of the server, it is worth explaining how Docker works behind the scenes. In this section, we will discuss the concepts of containers and how OpenWhisk works with the containers. Docker is an OS-virtualization software that contains Platform-as-a-Service (PaaS) products that are bundled in packages called containers and they are hosted in software called Docker Engine [20]. Containers are isolated by default, there are templates/images which can be downloaded from Docker Hub [21] and used directly or we can make personalized images and host them inside the containers. The containers can communicate with each

[19] Redis Sentinel: `https://redis.io/topics/sentinel`
[20] Docker Engine: `https://docs.docker.com/engine`
[21] Docker Hub: `https://hub.docker.com`

other via a pre-defined network. Docker provides handy CLI tools to manage the whole ecosystem, the documentation is vast and well defined [22]. There are various 3rd-party wrappers in almost every programming language for the CLI and can be integrated into the application seamlessly. We will be using dockernode [23] in our application to manage the docker containers for Redis. OpenWhisk creates several docker containers but mainly two of them are the core for its functioning, Invoker[24] and docker skeleton [25]. Invoker is responsible for invoking serverless functions and docker skeleton is the runtime container that can be considered as an isolated Blackbox for the supported languages by OpenWhisk.

4.4 Lambda

Lastly, we will discuss how AWS Lambda works, because we will be needing a benchmark for our results and we need a popular in-demand production-grade serverless cloud service. Similar to OpenWhisk, Lambda is an event-driven serverless service hosted by AWS, the provisioning and server management is handled by AWS. It follows the same concept of hot, warm, and cold startup of functions. It can be seamlessly integrated by mostly all other AWS services such as S3, SNS, SQS, API Gateway, etc. Internally, AWS Lambda uses Firecracker as mentioned in 2.6 to invoke isolated serverless functions with optimal performance and security.

Using the above technologies/frameworks we can design our model. To have a serverless Redis data store we will have to start the Redis container along with functions. Redis should be available when the functions are about to start, otherwise, the function will keep on trying in the loop and throwing connection errors. The lifecycle which we will be aiming for is shown in the Figure 6.

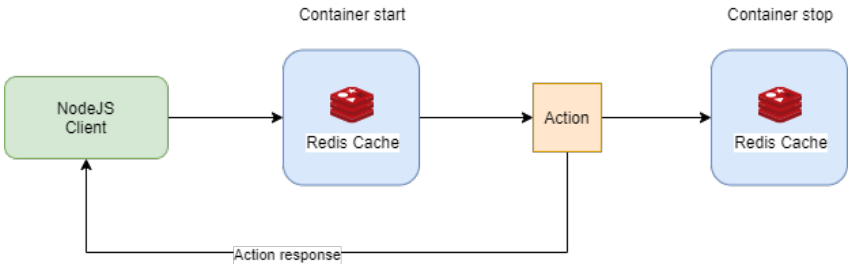

Figure 6: Experiment lifecycle

With the lifecycle figured out, we will be focusing on the model of our experiment. Our OpenWhisk model should have a client which will be managing the actions and docker container of Redis. Figure 7 represents the acrhitecture of the model.

5 Implementation

This section will be dedicated to the implementation of the code written to manage the experiment and operations in serverless functions. First, we will be going through the core of the experiment which

[22] Docker CLI: `https://docs.docker.com/engine/reference/commandline/cli`
[23] dockernode: `https://github.com/apocas/dockerode`
[24] OpenWhisk invoker image: `https://hub.docker.com/r/openwhisk/invoker`
[25] OpenWhisk docker skeleton image: `https://hub.docker.com/r/openwhisk/dockerskeleton`

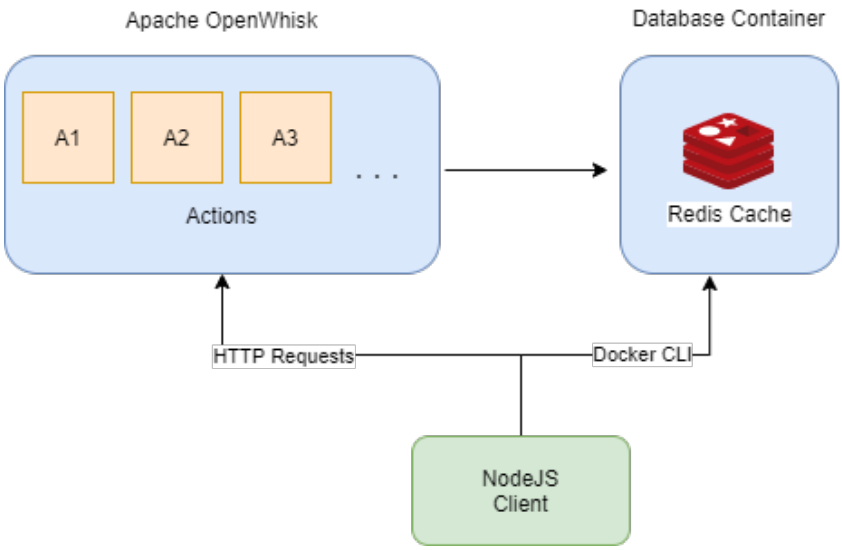

Figure 7: OpenWhisk and Redis container architecture

is responsible for starting the Redis Docker container, executing the action, and stopping the container after a certain time. The code is written in NodeJS and will be using third-party libraries as mentioned in 3.

To start the program via UI or code, we would require some parameters, apihost, api_key, name, kind, code e.g. code zip file path, and numberOfInvocation.

Execution flow will start with the Redis container, actions will be invoked, the actions will return JSON objects but the objects will not be returned via stream instead they will be stored in OpenWhisk logs. We will fetch the activations with the help of the name parameter as every activation JSON has a mandatory name object. Activation can easily be fetched via OpenWhisk CLI using `getActivation`function. To achieve serverless behavior in the Redis container we would like to save the database on the disk or remote place that could be fetched when we again start the container and shut the container after saving. In our experiment, we will be saving the data on the local disk. The activation data will be rendered on the UI where we will be adding the parameters.

In the action or serverless function, we are querying three operations on the Redis server. The operations will use an array of 1000 strings which will be generated by randomStringGenerator function written in the file. They will set, read, and delete the data referenced from the array. The operations will be synchronous e.g. each of them will be executing one-after-another. The code will be timing each operation with the help of the NodeJS native `performance` module[26].

Apart from the core program and actions, there is a UI component built with ExpressJS[27]. The express server will be listening on the index route for the parameters and serve the result data. We are using the HandlebarsJS templating engine[28] for rendering the view. For demonstration purposes, there will be two additional functions that will be able to showcase how the data can be shared among functions with the same Redis container. The functions will be executed one after the other, the first function will write some data to the database, wait for it to shut down, and the second function will

[26]NodeJS perf_hooks module: `https://nodejs.org/api/perf_hooks.html`
[27]ExpressJS: `https://expressjs.com`
[28]HandlebarsJS: `https://handlebarsjs.com`

retrieve the same data using the same key used in the first function in Redis. There will be a separate UI for these functions.

6 Evaluation

We have invoking three functions, first function is executed 100 times while recording the time of write, read, and delete operations of 1000 records on the database, the second and third function consists of code that evaluates the data sharing between two functions using this model. The first function is executed in AWS Lambda as well to compare the results with time metrics.

6.1 Case Study 1

The data we get is the time in milliseconds of every execution we have done so far, it includes the total time of function, read, write, and delete time of Redis data. As mentioned above we are operating on 1000 random strings on the database. The data shown below does not include the startup time of the Redis container which is approximately 2 seconds before 100 invocations of OpenWhisk actions.

First, we can see in Figure 8 the time taken by the functions to perform three operations e.g. write, read, and delete in both environments. There is a massive difference in time because Redis was deployed locally on the same server as OpenWhisk using Docker, whereas in the case of AWS Lambda the ElastiCache Redis could be far from the functions container. Even Lambda and ElasticCache shared the same VPC the response time of ElastiCache is much higher. Table 2 represents the mean time and standard deviation of all three operations.

Table 2: AWS Lambda vs OpenWhisk: Function operations time (ms)

	OpenWhisk	**AWS Lambda**
Write (Mean (ms))	178.56	3136.24
Write (Std Dev (ms))	13.27	945.55
Read (Mean (ms))	168.78	372.56
Read (Std Dev (ms))	11.89	2050.89
Delete (Mean (ms))	9.76	165.81
Delete (Std Dev (ms))	2004.56	459.86

As mentioned before, we also got the overall duration of the function which includes initializing libraries and language runtime, along with the Redis operations. This is obvious from the previous graphs that the overall time of the AWS Lambda functions is higher than OpenWhisk actions. The Figure 9 shows the comparison of time in both cases and Table 3 represents the mean and standard deviation of the time.

Table 3: AWS Lambda vs OpenWhisk: Overall mean and standard deviation

	Mean (ms)	**Standard deviation (ms)**
OpenWhisk	538.93	148.93
AWS Lambda	6394.37	2894.35

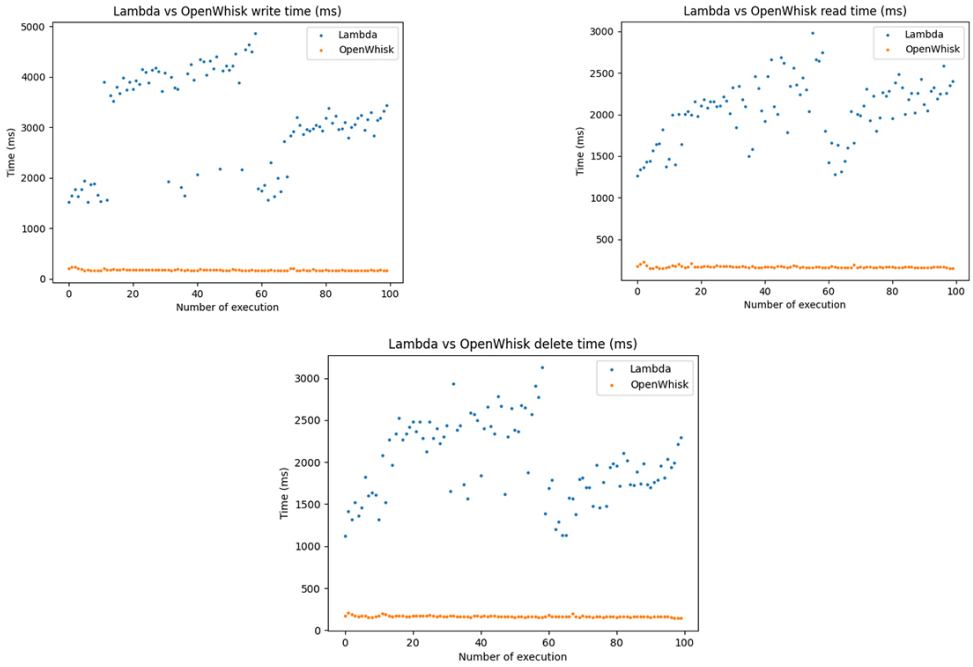

Figure 8: AWS Lambda vs OpenWhisk: Write, Read and Delete operations time (ms)

In the Figure 9, it is clear that AWS Lambda response time is not consistent, because of the availability of ElastiCache. There are some functions that returned the values at almost the same time as OpenWhisk, as these durations are not counted with cold starts of the function we can safely assume that these time variations are caused by incosistent connection between Lambda and ElastiCache.

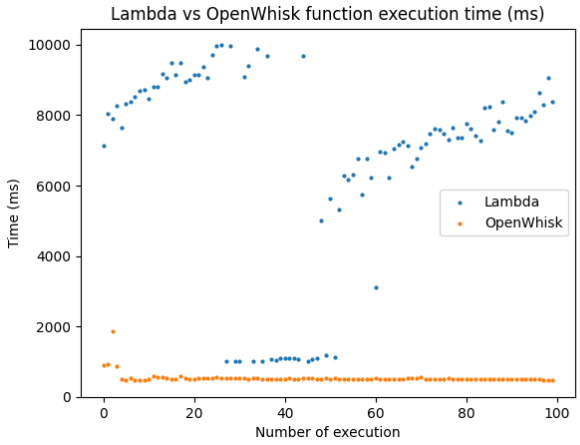

Figure 9: AWS Lambda vs OpenWhisk: Overall function execution time (ms)

6.2 Case Study 2

In this case study, we used two functions to share data using Redis. The two functions are very simple but enough to evaluate the data-sharing capability of the model. The first function sets the data in the

Redis container, waits for some time until the container is shut-off, and then invokes the second function which will retrieve the same data from Redis. The data saved was studentID and name. The experiment was conducted 20 times with 100% success rate.

6.3 Discussion

We can see a vast difference between function execution time, OpenWhisk with local Redis performs approximately 85% better than AWS Lambda and ElastiCache. But there are several differences that are responsible for the result such as ElastiCache is a different service that may reside in a different server altogether in the same AWS data warehouse, but technically they should not have high latency as they share the same VPC, as shown in Table 1, ElastiCache has dedicated server that does not share any resource with any other service that similarly with Lambda, so there should not be any compute bottleneck and we can safely assume that compute was not the reason to have higher execution time. For 100 executions of AWS Lambda functions, we get an average of 6.3 seconds that is still higher even if we consider startup time for Redis container for every function execution in OpenWhisk. Because the average for OpenWhisk functions is 538 ms and if we add 2 seconds of Redis startup that will still be less than the AWS Lambda average. The limitation of this model would be the boot time of the Redis container will delay the cold-start of the function even more. If an application requires a quick response from the functions and the functions are used rarely, using ElastiCache like services will be much preferable, though it may cost more.

7 Conclusion and Future Work

The experiments show that the OpenWhisk and Docker Redis implementation performs better than the AWS Lambda and ElastiCache stack by a hefty margin. But there are some limitations with the current implementation, it is useful for those applications which require bulk serverless functions as there is a time penalty for starting and stopping the Redis container if the functions are invoked rarely then more cost and time will be involved. The functions and container should preferably reside in the same machine or nearby otherwise there will be latency issues similar to AWS Lambda and ElastiCache's current implementation.

In the current implementation, to achieve complete persistence and serverless behavior we could save the Redis data using one of its saving strategies e.g. RDB into the cloud storage like S3, and retrieve the data back when starting the Redis container, there will be high latency involved but it will still cost less than ElastiCache. ElastiCache's seed data option is similar to this approach [29]. We can scale the Redis using containers and Kubernetes depending on the data and usage of the connected application.

References

Abad, C. L., Boza, E. F. & van Eyk, E. (2018), 'Package-aware scheduling of faas functions', *ICPE 2018 - Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*

[29] ElastiCache S3 seed: `https://pages.awscloud.com/rs/112-TZM-766/images/Session%201%20-%20ElastiCache-DeepDive_v2_rev.pdf`

2018-January, 101–106.

Agache, A., Brooker, M., Florescu, A., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P. & Popa, D.-M. (2020), *Firecracker: Lightweight Virtualization for Serverless Applications*.
URL: *https://www.usenix.org/conference/nsdi20/presentation/agache*

Ahamed, W. S. S., Zavarsky, P. & Swar, B. (2021), 'Security audit of docker container images in cloud architecture', *ICSCCC 2021 - International Conference on Secure Cyber Computing and Communications* pp. 202–207.

Baldini, I., Castro, P., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R. & Suter, P. (2016), 'Cloud-native, event-based programming for mobile applications', *Proceedings - International Conference on Mobile Software Engineering and Systems, MOBILESoft 2016* pp. 287–288.
URL: *https://www.researchgate.net/publication/305524856_Cloud-native_event-based_programming_for_mobile_applications*

Bhattacharjee, A., Barve, Y., Khare, S., Bao, S., Gokhale, A. & Damiano, T. (2019), 'Stratum: A serverless framework for lifecycle management of machine learning based data analytics tasks'.
URL: *https://www.researchgate.net/publication/332186165_Stratum_A_Serverless_Framework_for_Lifecycle_Management_of_Machine_Learning_based_Data_Analytics_Tasks*

Ghosh, B. C., Addya, S. K., Somy, N. B., Nath, S. B., Chakraborty, S. & Ghosh, S. K. (2020), 'Caching techniques to improve latency in serverless architectures', *2020 International Conference on COMmunication Systems and NETworkS, COMSNETS 2020* pp. 666–669.

Klimovic, A., Litz, H. & Kozyrakis, C. (2017), 'Reflex: Remote flash local flash', *ACM SIGPLAN Notices* **52**, 345–359.
URL: *http://dx.doi.org/10.1145/3037697.3037732*

Klimovic, A., Wang, Y., Kozyrakis, C., Stuedi, P., Pfefferle, J. & Trivedi, A. (n.d.), 'Understanding ephemeral storage for serverless analytics'.
URL: *https://www.usenix.org/conference/atc18/presentation/klimovic-serverless*

Klimovic, A., Wang, Y., University, S., Stuedi, P., Trivedi, A., Pfefferle, J. & Kozyrakis, C. (2018), 'Pocket: Elastic ephemeral storage for serverless analytics pocket: Elastic ephemeral storage for serverless analytics'.
URL: *https://www.usenix.org/conference/osdi18/presentation/klimovic*

Lee, H., Satyam, K. & Fox, G. (2018), 'Evaluation of production serverless computing environments', *IEEE International Conference on Cloud Computing, CLOUD* **2018-July**, 442–450.

Mocanu, G., Carabas, C. & Tapus, N. (2021), 'Fuzz testing in aws firecracker hypervisor', *Proceedings - 2021 20th International Symposium on Parallel and Distributed Computing, ISPDC 2021* pp. 130–137.

Preeth, E. N., Mulerickal, J. P., Paul, B. & Sastri, Y. (2016), 'Evaluation of docker containers based on hardware utilization', *2015 International Conference on Control, Communication and Computing India, ICCC 2015* pp. 697–700.

Quevedo, S., Merchan, F., Rivadeneira, R. & Dominguez, F. X. (2019), 'Evaluating apache openwhisk - faas', *2019 IEEE 4th Ecuador Technical Chapters Meeting, ETCM 2019* .

Yadav, R., Zhang, W., Li, K., Liu, C., Shafiq, M. & Karn, N. K. (2018), 'An adaptive heuristic for managing energy consumption and overloaded hosts in a cloud data center', *Wireless Networks 2018 26:3* **26**, 1905–1919.
URL: *https://link.springer.com/article/10.1007/s11276-018-1874-1*

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S. & Stoica, I. (2012), *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*.
URL: *http://ec.europa.eu/eurostat/statistics-explained/index.php/International*$_t rade_i n_g oods$