FACULTY OF ENGINEERING AND TECHNOLOGY

MASTER OF SOFTWARE ENGINEERING

MASTER THESIS

Investigate the Performance of Serverless Functions Deployed on the Cloud Computing Environment Using Different Containers Orchestration Frameworks

Author:
Mohammed AbuAisha
(1175033)

Supervisor:
Dr. Yousef Hassouneh

September 11, 2020

This thesis was submitted in partial fulfillment of the requirements for the Master's Degree in software engineering from the Faculty of Graduate Studies, at Birzeit University, Palestine

Investigate the Performance of Serverless Functions Deployed on the Cloud Computing Environment Using Different Containers Orchestration Frameworks

Author
Mohammed AbuAisha

This thesis was prepared under the supervision of Dr. Yousef Hassouneh and has been approved by all members of the examination committee

Dr. Yousef Hassouneh, Birzeit University
(Chairman of the Committee)

[illegible]

Dr. Abdel Salam Sayyad, Birzeit University
(Member)

[illegible]

Dr. Majed Ayyad, Birzeit University
(Member)

[illegible]

Date of Defense:
August 22, 2020

Declaration of Authorship

I, Mohammed AbuAisha, declare that this thesis titled, “Investigate the Performance of Serverless Functions Deployed on the Cloud Computing Environment Using Different Containers Orchestration Frameworks” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research de-gree at Birzeit University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Birzeit University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed my-self.

Signed: Mohammed AbuAisha

Date: 09 / 11 / 2020

Abstract

Serverless computing became an interesting topic recently in the field of software industry because of the benefits that brings to the organizations of managing and deploying complex software that requires heavy computations. Many public cloud providers offer Serverless services that can cope with organization needs for managing and handle heavy load software applications where performance is a critical factor. That leads to many studies in academia that tackled the performance of Serverless in public cloud. However, the idea of applying Serverless using open source frameworks became more popular to address the limitations of using Serverless provided by public cloud providers specially for Internet of things, edge computing applications which attracted researchers to study the performance of Serverless. In this work, we studied the performance in terms of response time, throughput and success rate using container orchestrators to investigate their impacts on Serverless functions where we evaluated 9 open source Serverless frameworks and did a comparisons between them to select the best candidate for our study. The OpenFaas Serverless framework was mainly selected based on its support for running under different container orchestrators and ease of installing and configuring it among others that support different container orchestrators. We conducted the experiment of this study using a custom tool called *faas-exp* that we built for this purpose that facilitated infrastructure provisioning, configuration management, automation of test cases generation, data analysis's and visualization. Three container orchestrators were used on this study Docker Swarm, kubernetes and Nomad in order to find the impact of container orchestrator for the deployed Serverless functions. Our results showed that there is a relationship between the container orchestrator and the performance of deployed Serverless functions based on different computation requirements, programming languages/runtimes and warm & cold start.

الملخص

أصبحت الحوسبة بدون خوادم موضوعًا مثيرًا للاهتمام مؤخرًا في مجال صناعة البرمجيات بسبب ما تقدمه للشركات من فوائد تسهل عملية إدارة وتهيئة البرامج الضخمة التي تتطلب عمليات حسابية معقد . يقدم العديد من مزودي الخدمات السحابية العامة خدمات بدون خادم يمكنها التعامل مع احتياجات الشركات من أجل الإدارة والتعامل مع البرمجيات التي تقدم خدمات لعدد كبير من المستخدمين حيث يعتبر الأداء عاملاً بالغ الأهمية.بالمقابل دفع ذلك الكثير من الباحثين إلى دراسة والتحقق من أداء الدوال من دون خادم على الخدمات السحابية العامة. من ناحية أخرى أصبحت فكرة دراسة الدوال من دون خادم باستخدام أطر من دون خادم مفتوحة المصدر شائعه من أجل معالجة وتخطي المشاكل الناتجة عن استخدام الحوسبة بدون خوادم المقدمة من قبل مزودي الخدمات السحابية العامة خصوصا في تطبيقات إنترنت الأشياء و حوسبة الحافة التي جذبت اهتمام الباحثين لدراسة والتحقق من أداء الدوال من دون خادم باستخدام هذه الأطر. لقد قمنا في هذا البحث بدراسة الأداء من حيث وقت الاستجابة,الإنتاجية و معدل النجاح باستخدام أطر الأوعية المتزامنة المختلفة للتحقيق في تأثيرهم على وظائف الدوال من دون خادم حيث قمنا بعمل مقارنة وتقييم لأطر دوال من دون خادم مختلفة من أجل اختيار أفضل إطار دوال من دون خادم يتناسب مع هدف الدراسة.تم اختيار إطار دوال من دون خادم (اوبن فاس) بشكل أساسي بناءً على دعمه أطر أوعية متزامنة مختلفة لسهولة تنصيبه وتهيئته مقارنه بأطر اخرى تدعم أطر أوعية متزامنة مختلفة. لقد قمنا بإجراء تجربة لغاية هذا البحث باستخدام أداة تسمى (فاس اكسب) قمنا بتطويرها لهذا الغرض والتي تسهل إنشاء البنية وإدارة التهيئة , إنشاء حالات الاختبار,وتحليل البيانات إنشاء الرسوم البيانية.تم استخدام ثلاثة أطر أوعية متزامنة مختلفة في هذه الدراسة (دوكر سوارم) و (كوبرنيتس) و (نوماد) من اجل ايجاد دليل على تأثير أطر أوعية متزامنة مختلفة على أداء دوال من دون خادم , وأظهرت نتائجنا أن هناك علاقة بين أطر أوعية متزامنة مختلفة وأداء دوال من دون خادم المنتشرة بناءً على متطلبات حسابية مختلفة , لغات البرمجة والأوقات المختلفة.

Acknowledgements

Praise be to God that helped me to accomplish and finish this dissertation. This research could not be performed without help and support from my advisor Dr. Yousef Hassouneh for monitoring me through the period of working on this thesis.

Finally I would like to extend my deepest gratitude to my parents and wife for supporting, helping and understating me to accomplish this achievement where I could never have completed this without them.

Abbreviations and Acronyms

API	Application Programming Interface.
Async	Asynchronous.
AWS	Amazon Web Services.
cgroups	Control Groups.
CLI	Command Line Interface.
CRUD	Create, Read, Update, Delete.
FaaS	Function as a Service.
GCP	Google Cloud Platform.
HTTP	HyperText Transfer Protocol.
IaaS	Infrastructure as a Service.
IoT	Internet of Things.
JS	JavaScript.
JVM	Java Virtual Machine.
NIST	National Institute of Standards and Technology.
PaaS	Platform as a Service.
RESTful	Representational state transfer.
SaaS	Software as a Service.

Sync	Synchronous.
VM	Virtual Machine.
VMM	Virtual Machine Monitor.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Software architecture is the main stage in software development life cycle, it helps to build software that meets both business objectives and technical requirements. Software architecture depicts the structure of the software, the interaction between software components and external entities. It also shows the deployment of the software. In the past, the monolithic architecture was a well known approach to build a software in which all of its components combined into a single piece where everything is developed and deployed as a single artifact [112]. It affects the scalability which requires to deploy the whole software when need more resources to scale. Moreover, the whole software will be affected if one of its components might be failed. The high coupling between components will affect the whole software and affects the scaling efficiency and availability. Aforementioned can be avoided using different approaches. Microservice is one of the approaches which decomposes the software into multiple services where each service has single responsibility and well-defined boundaries, integrated with lightweight and general purpose communication protocols to build a software [97]. In the past, software deployment requires from developer to purchase bare metal machines and either installed them on-premise or leased rack space in the data center [104]. Moreover, they had to over-provision their infrastructure to account for scalability and resilience [110]. Using bare metal solution or leased rack space cause to leave un-utilized resources for a considerable amount of time. The rising of hardware virtualization improves the level of resource utilization outstandingly using hypervisor-based virtualization which gives the ability to host multiple virtual machines and run them in top of a single physical machine that helps to deploy software across different virtual machines. This also allows to increase resource utilization for single physical machine, create significant cost savings, simplify IT management, eliminate downtime, provision and deprovision resources on demand

and improve business continuity and disaster recovery. Virtualization opens the door to pay more attention to the cloud computing to get better performance for sensitive workload management of software services [109]. The elasticity provided by the virtualization introduces a new cloud computing model IaaS (Infrastructure as a Service) where new VMs can be provisioned in order to cope with increased workloads [106]. Moreover, another model arises known as Platform-as-a-Service (PaaS) which gives developers the flexibility for implementing, executing, and managing software without the complexity of managing and maintaining the underlying infrastructure. Container-based virtualization is a new type of virtualization that becomes more popular recently which provides software on demand. The stability and maturity of such technologies encourage developers to shift building software based on monolithic architecture and decompose them into small pieces of software based on distributed architecture using microservices. New cloud computing paradigm (Serverless) has been arised because of highly adoption and usage of containers and microservices architecture. Serverless computing has many definitions and it is one of the cloud computing models which gives the ability to run event-driven and granularly billed software without needing to address the operational logic [92] and it will be a suitable architecture for workloads which have sporadic demands and for workloads that are short, asynchronous or event-driven and concurrent [90]. Serverless shares some of the characteristics of microservices but it is more granular than microservices, and provides a much higher level of functionality. On the other hand, Function-as-a-Service (FaaS) is a form of Serverless computing where the cloud provider manages the resources, lifecycle, and event-driven execution of user-provided functions [92]. Serverless computing is offered and provided by most of the well known public cloud providers like AWS, Azure and GCP who offers Serverless computing on their infrastructure. Developers can deploy their code in the form of function to the cloud provider, and the provider is responsible for the execution, resource provisioning and automatic scaling of the runtime environment [93]. It also helps to cut operational cost and facilitate resource management and utilization. Most of the Serverless computing offered by public cloud providers package function in a stateless container that helps developers to focus on code without need to worry about managing the underlying infrastructure, auto-scaling and paying only for usage. However, the public cloud platforms have certain limitations related to vendor lock-in and restrictions on the computation of the functions [105]. To cope with these limitations multiple open source Serverless frameworks are available where Serverless can be deployed to on-premise or even to public cloud infrastructure. Since Serverless function can be deployed as a container, then using container orchestration can be useful to deploy and manage functions. Recently, there is an increasing interest to implement

Serverless on a on-premise infrastructure using container orchestrations.

1.1 Motivation

Building and deploying software requires huge time and effort to release it to the market but how much time is required to manage it ?. This is where a new cloud computing paradigm called Serverless came to the surface, because it abstracts the underlying operating systems and infrastructure. It does not mean there are no servers, but it's the responsibility of the cloud provider to allocate the resources and developer does not need to worry about the server management. Moreover, it takes care of resource scaling automatically based on software load. Organizations pay only for the resources usage which means when the software really harness the resources, they do not need to pay anything if software functions are not using any resources where it explains the increasing adoption of Serverless computing which provided by well known public cloud providers. However, regardless of the great benefits provided by public platforms but there are some limitations need to be considered when it comes to using public cloud. The multitenancy problem can affect system performance, security concerns and robustness. Vendor lock-in is another serious problem since each cloud provider also has limitations in terms of programming languages supported, maximum execution duration, maximum concurrent executions and so on [104]. All these limitations justifies the movement of organization toward using Serverless on on-premise infrastructure where they can have full control over the infrastructure running the Serverless. As new technologies are emerging such as smart home devices, self driving cars and augmented reality require new approaches to deal with the network traffic generated by the IoT devices to enable such technologies [86]. Moreover, Edge computing play crucial roles in accelerating data streaming for processing real-time data with low latency. On the other hand, it helps smart software and devices to respond to data almost instantaneously, as its being created, eliminating lag time [88]. Using Serverless on on-premise infrastructure can be incorporated at the edge of an IoT network which helps to execute small tasks and reduce latency. With Serverless it is possible to deploy functions to the edge network where each function can be triggered in response to an event. It is the responsibility of a Serverless framework to handle auto-scaling for workloads that can varies from zero to thousands of instances of function which helps to improve resources utilization. Aforementioned, and based on increasing interests to study Serverless on on-premise infrastructure and the ability to deploy function as container encouraged us to start studying this topic. Adoption of Serverless on on-premise infrastructure can be done using containers. Moreover, there are multiple implementations for open source orchestration frameworks that can help to deploy, manage and scale Serverless. The following are good examples of open source frameworks: Kubernetes, Docker Swarm, Nomad and Apache Mesos. The goal of this thesis is to evaluate the performance of the Serverless functions that support running under different container orchestrators. Also, the effect of the container orchestrators on the Serverless functions will be investigated and measured under different configurations. The public cloud provider AWS is used in this work as our infrastructure to investigate our research.

1.2 Research Questions

The main goal of this is to measure the performance of Serverless functions under different container orchestrators and investigate if such orchestrators can affect the deployed functions, so that the following research questions need to be answered

1. **RQ1)** What is the impact of container orchestrators on the Serverless functions performance ?.
   (a) **RQ1.1** What is the impact of container orchestrators on response time of Serverless function ?.
   (b) **RQ1.2** What is the impact of container orchestrators on throughput of Serverless function ?.
   (c) **RQ1.3** What is the impact of container orchestrators on success rate of Serverless function ?.
2. **RQ2)** How the performance of Serverless function is affected under different workload requests ?.
   (a) **RQ2.1** How response time of Serverless function affected under different workload requests ?
   (b) **RQ2.2** How throughput of Serverless function affected under different workload requests ?
   (c) **RQ2.3** How success rate of Serverless function affected under different workload requests ?
3. **RQ3)** How various computational requirements affect Serverless functions performance ?.
   (a) **RQ3.1** How various computational requirements affect response time of Serverless function ?.
   (b) **RQ3.2** How various computational requirements affect throughput of Serverless function ?.
   (c) **RQ3.3** How various computational requirements affect success rate of Serverless function ?.

4. **RQ4)** What is the effect of the runtimes/programming languages on the Serverless functions performance ?.
   (a) **RQ4.1** What is the effect of using runtimes/programming languages on response time of Serverless function ?.
   (b) **RQ4.2** What is the effect of using runtimes/programming languages on throughput of Serverless function ?.
   (c) **RQ4.3** What is the effect of using runtimes/programming languages on success rate of Serverless function ?.
5. **RQ5)** What is the impact of COLD and WARM requests on the Serverless functions performance ?
   (a) **RQ5.1** What is the impact COLD and WARM requests on response time of Serverless function ?.
   (b) **RQ5.2** What is the impact COLD and WARM requests on throughput of Serverless function ?.

1.3 Structure of thesis

The rest of this thesis is structured as follows. Chapter 2 introduces background about virtualization, cloud computing, containers, microservices, container orchestration frameworks, Serverless computing and open source Serverless frameworks. Chapter 3 discusses related work in the field of Serverless and focuses on the limitations and gaps for the extant researches in Serverless computing. Chapter 4 provides full details about the research methodology conducted to achieve the goal of the research work. Chapter 5, presents the experiment results. Chapter 6 presents the discussion about the experiment results. Finally, Chapter 7 provides conclusion, threats to validity and future works.

Chapter 2

Background

The main goal of this chapter is to provide a basic background for some terminologies and concepts that will be used during this thesis. Section 2.1 discusses the mircroservices architecture. Section 2.2 presents virtualization concepts. Section 2.3 provides basic information about the cloud computing and its different models. Section 2.4 introduces Docker as Linux Container. Section 2.5 discusses container orchestrations for the selected platforms used in this research. Section 2.6 discusses the Serverless computing. Finally, Section 2.7 discusses about the open source Serverless frameworks

2.1 Microservices

Microservice architecture is one of the important topic in the field of software design which has been getting a lot of attention among software developers and researchers [123]. Companies like Netflix is one of the earlier immigrant of software from monolithic to microservice architecture. Thus they moved their entire systems by decomposing them into hundreds of microservices based on cloud computing. Microservice is a software architecture that is based on well-defined level of modularization, in which software can be modularized as small set of services. Each service is implemented and operated as an independent software, that allows communication with other services using lightweight protocol. The modularity provided by microservice increases software agility [96] as each service can be developed, deployed, operated and scaled independently. Figure 2.1 shows how microservice and monolithic architecture can be represented. Monolithic architecture wraps multiple services and run them as single deployable. However, microservice decomposes the software into small, independent services that can communicate using lightweight protocol. The

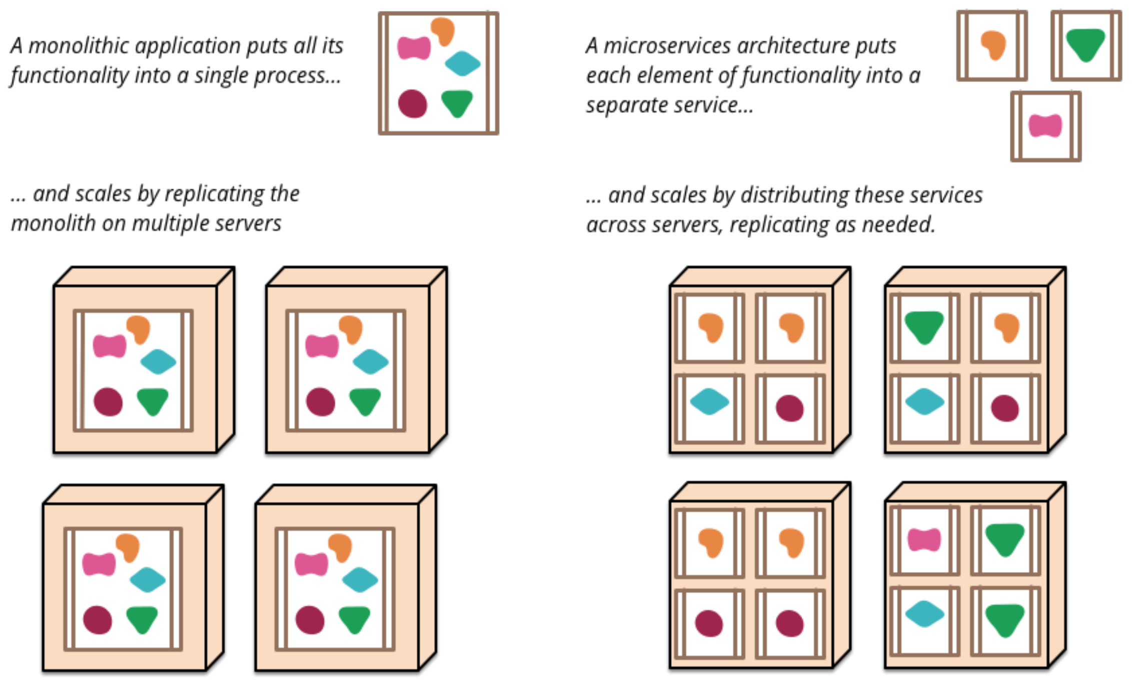

A monolithic application puts all its functionality into a single process...
A microservices architecture puts each element of functionality into a separate service...
... and scales by replicating the monolith on multiple servers
... and scales by distributing these services across servers, replicating as needed.

Figure 2.1: Microservice and Monolithic Architecture

Scaling on monolithic architecture requires a redundant copies of the same software since the the whole software should be scaled as one unit which causes extra unused hardware. Moreover, as everything tied together, the modification process becomes complex even for small changes which requires to redeploy the whole software and any failure in one of the services could cause failure to the whole software. Monolithic architecture forces technology lock-in as everything built in one unit which requires to keep using same technology, language and framework. On the other hand, microservice provides more advantages over monolithic. First, scaling microservice requires only to scale the services need to be scaled. Second, the whole software will be operational when one of the software services fails. Third, small changes is much easier and will not introduce extra errors. Fourth, deployment of new changes do not require to re-deploy the whole software as it only deploys the needed services. Finally, microservice depends on technology heterogeneity, which allows each service to use different technology than the other services to achieve the desired goals and performance [91].

2.2 Virtualization

Virtualization gains more attraction in the recent years and is considered the cornerstone of cloud computing. It allows single physical machine to run multiple virtual instances on top of it where each instance is isolated, scaled, operated independently from others. Moreover, it helps in resource computing utilization and cost reduction. The credits goes to IBM for virtualization in 1960's when they presented the idea of M44/44X system [122]. Hypervisor-based virtualization is one of the most common virtualization techniques where Xen [1], VMware [2] and KVM [3] are examples for hypervisor-based virtualization. Hypervisor is a software which has a full control of the underlying physical machine where it helps to manage the virtual machines using Virtual Machine Monitor VMM component where it implements the VM hardware abstraction, partitions and resource sharing that includes CPU, memory, and I/O for enabling virtualization of the underlying physical machine [118]. Figure 2.2 illustrates the Hypervisor-based virtualization architecture where each VM is isolated from each other and run its own operating system. This gives the underlying physical machine the ability to run multiple operating systems. On the other hand, Container-based virtualization is an alternative to the hypervisor since it provides a lightweight virtualization layer on the operating system level where it allows to create multiple isolated user-space instances on top of the same OS kernel [120]. Guest processes running inside container are isolated from other containers because of the abstraction provided by containers on top of the OS kernel. Linux-VServer [4], OpenVZ [5] and Linux Containers (LXC) [6] are implementations of Container-based virtualization. Figure 2.3 shows the difference between Container-based and Hypervisor-based virtualization.

[1]Xen: `https://xenproject.org/`

[2]VMware: `https://www.vmware.com/`

[3]KVM: `https://www.linux-kvm.org/`

[4]Linux-VServer: `http://linux-vserver.org/`

[5]OpenVZ: `https://openvz.org/`

[6]LXC: `https://linuxcontainers.org/`

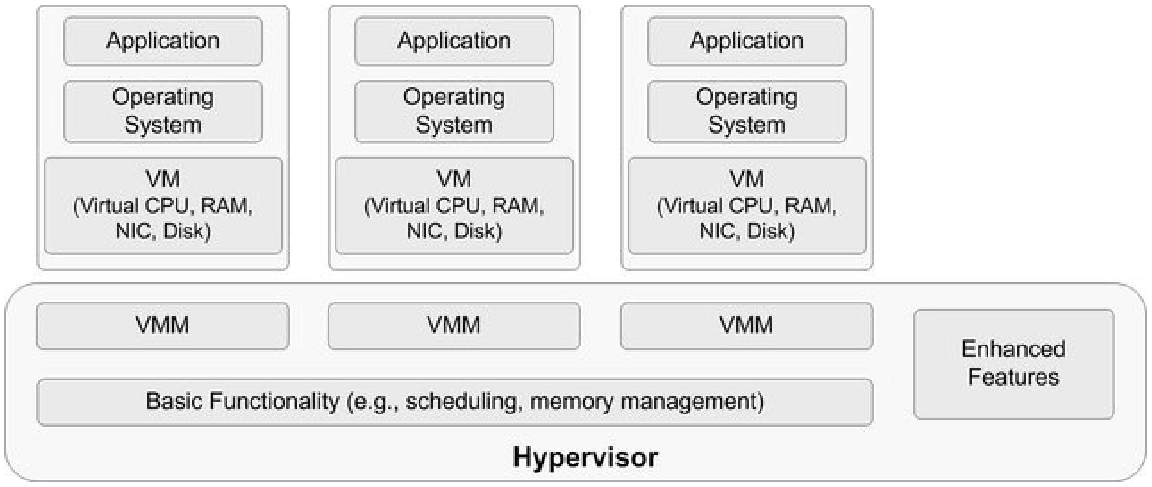

Application
Application
Application
Operating System
Operating System
Operating System
VM (Virtual CPU, RAM, NIC, Disk)
VM (Virtual CPU, RAM, NIC, Disk)
VM (Virtual CPU, RAM, NIC, Disk)
VMM
VMM
VMM
Enhanced Features
Basic Functionality (e.g., scheduling, memory management)
Hypervisor

Figure 2.2: Hypervisor-based Virtualization Architecture [118]

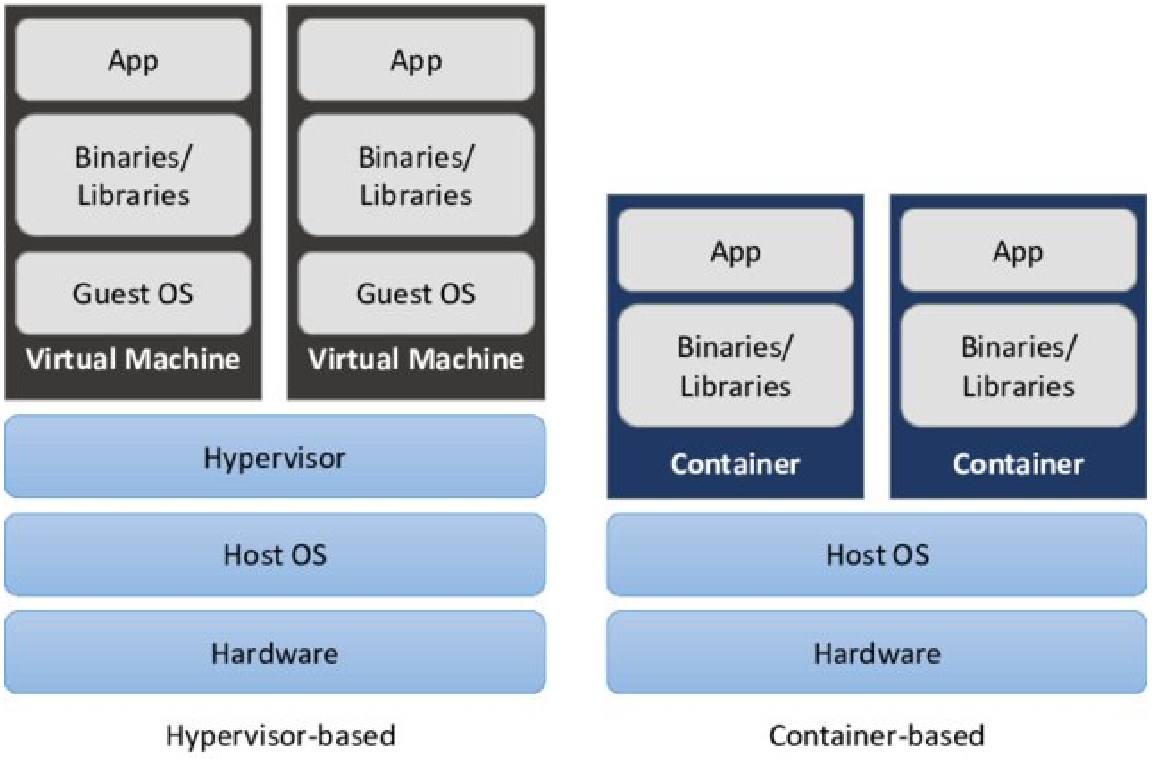


Figure 2.3: Hypervisor-based vs Container-based [115]

2.3 Cloud Computing

Cloud computing becomes so popular in IT industries recently. The NIST (National Institute of Standards and Technology) defines cloud computing as a model which enables on-demand network access to a shared pool of computing resources that includes servers, storage, networks, services and applications that can be easily provisioned and destroyed without management complexity or service provider interaction [71]. Cloud computing frees users from requirements of configuring and building their own infrastructure and allows them to pay only for the resources they used which reduces the cost significantly. Moreover, it provides the flexibility for scale in/out the infrastructure to fulfill user requests. On the other hand, it provides security and protection techniques which includes secure connection, data encryption, key management, and security threat monitoring.

Cloud computing provides three different models that can be divided into Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS).

1. **Software as a Service (SaaS)**: A model where the services provided to the user deployed over the Internet without needing to install or configure any hardware or software, the only requirement is to be able to access the internet. Service provider can charge users using subscription license ("pay-as-you-go") model or make the services free without charging if there is a chance to generate revenue using advertisements. [121]. Gmail, Outlook and Google Docs are example of SaaS services.

2. **Platform as a Service (PaaS)**: A model where service provider delivers a development environment and the whole solution stack for users as a service which allows them to deploy their own software on the provider infrastructure without worrying about complexity of the underlying hardware and software layers. Service provider provides all of the tools required to support the full life cycle of building, deploying and delivering software to the end users over the internet without needing to downloads or install anything for developers, IT managers or even the end users. Heroku [7], AWS Elastic Beanstalk [8] are examples of PaaS services.

3. **Infrastructure as a Service (IaaS)**: A model where service provider delivers servers, storage, network, and operating system as service to the end users.

[7]Heroku, `https://www.heroku.com/`

[8]AWS Elastic Beanstalk: `https://aws.amazon.com/elasticbeanstalk/`

Users can use the infrastructure provided by service provider to create multiple virtual machines upon needs without worrying about purchasing any servers, network and storage equipment to build their own datacenter. AWS [9], GCP [10] and Azure [11] are examples of IaaS services.

Cloud computing can be classified into 3 main classes as the following:

1. **Public Cloud**: The underlying computing infrastructure is provided by service provider datacenters distributed across multiple locations over the world where user knows nothing about the exact location of the cloud computing infrastructure and the infrastructure is shared between all users. The services provided by cloud computing can be accessed and available for any user who wants to subscribe and uses them by paying only for the actual resources usage.

2. **Private Cloud**: The underlying computing infrastructure is reserved and dedicated only for a user and cannot be shared with others. Private Cloud is more expensive and secure than Public Cloud.

3. **Hyprid Cloud**: Hybrid cloud combines public cloud and private cloud where private cloud responsible for hosting sensitive software data and whereas public cloud can be used for non sensitive software. Each cloud can be managed independently from each others.

2.4 Docker

Docker [12] is an open source platform which helps developers to build, ship and deploy their software and package them into standard format called containers which contains all the required dependencies for that software [114]. It also helps to isolate resources using features provided by the Linux kernel where it depends on cgroups and namespaces. cgroups allows hardware resources sharing and make them available for containers and force them to use specific amount of resources such as memory or CPU. Moreover, docker creates multiple namespaces for each container that are related to process, network, file-system and inter process communication which helps to isolate containers from each others [74]. Docker provides the flexibility for developers to focus on developing their software without worrying about the operating system where software is running.

[9]AWS: `https://aws.amazon.com/`

[10]GCP: `https://cloud.google.com/`

[11]Azure: `https://azure.microsoft.com/en-us/`

[12]Docker: `https://www.docker.com/`

Docker has 5 main components which includes Docker daemon, Docker client, Docker registry, Docker images and Docker containers as specified in Figure 2.4.

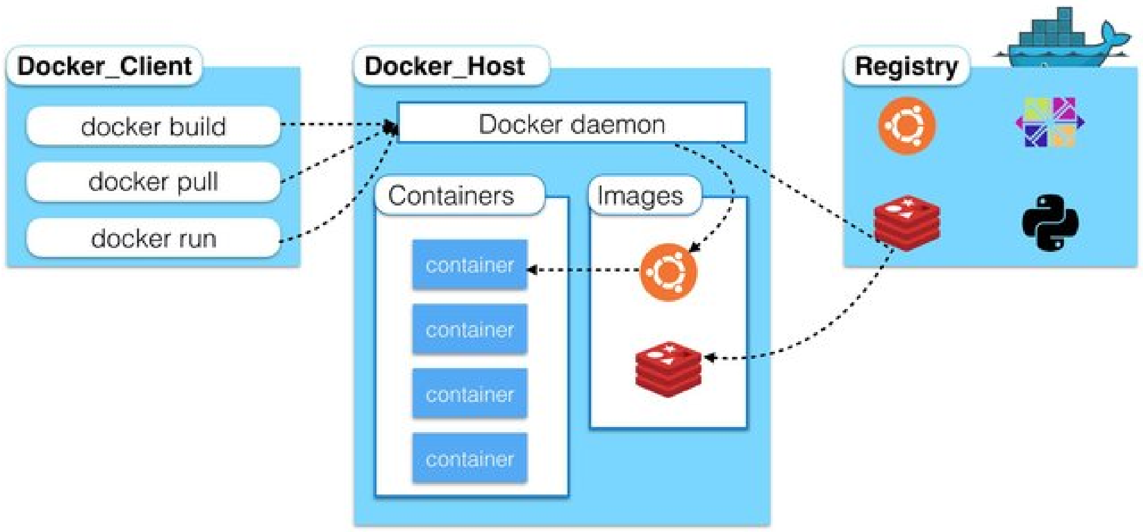


Figure 2.4: Docker Architecture [117]

1. **Docker Daemon**: The Docker daemon called dockerd which represents the docker RESTful API that accepts API requests. It also helps to manage and create different types of docker objects which includes images, containers, volumes, and networks.

2. **Docker Client**: A CLI client tool allows users to interact with Docker daemon by running different commands for managing docker objects stored at the docker server side.

3. **Docker Registry**: A place where docker images is stored. Docker Hub [13] is an example of public docker registry where everyone can access any docker image they want. It also gives an option to create private repositories for storing docker images. Moreover, there are multiple private docker registries other than Docker Hub provided by other service providers which includes: AWS ECR [14], JFrog Container Registry [15] and many others.

[13]Docker-Hub: `https://hub.docker.com/`
[14]AWS ECR: `https://aws.amazon.com/ecr/`
[15]JFrog: `https://jfrog.com/container-registry/`

4. **Docker Images**: A Read-only template which contains instructions for how to build containers. There is a base image for every docker image where operating systems are the base images for any image that need to be created.

5. **Docker Containers**: A running instance of docker image called container. Container encapsulates the whole binaries and requirements for the running software [114].

2.5 Container Orchestrations

Container enables fast deployment with low overhead and provides high degree of scalability for software based on microservice architecture [80]. Adopting container solution for complex and large software becomes challenging and difficult as the number and interdependencies of containers increases. This explains the need for container management layer known as container orchestration to help provisioning and managing containers by focusing on scheduling, resource allocation, scaling, load balancing, monitoring and exposing services to the outside world. Kubernetes [16], Docker Swarm [17], Nomad [18], Mesos [19] and OpenShift [20] are examples of container orchestrator frameworks. In this research we are going to focus only in Kubernetes, Docker Swarm and Nomad.

2.5.1 Kubernetes

Kubernetes is an open source platform developed by Google and licensed under Apache 2.0. Google started using container technology internally around 20 years ago by building Borg system which is a cluster manager responsible for running hundreds of thousands of jobs that hides the complexity of resource management and failure from developers to let them focus on their code [119]. Google developed an open source version of Borg called Kubernetes which is considered to be one of the most popular container orchestrator platforms. Figure 2.5 shows the Kubernetes Architecture.

Deploying Kubernetes produces cluster where it contains at least one master node and one worker node. Kubernetes can be divided into two main parts: master

[16]Kubernetes: `https://kubernetes.io/`
[17]Docker Swarm: `https://docs.docker.com/engine/swarm/`
[18]Nomad: `https://www.nomadproject.io/`
[19]Mesos: `http://mesos.apache.org/`
[20]OpenShift: `https://www.openshift.com/`

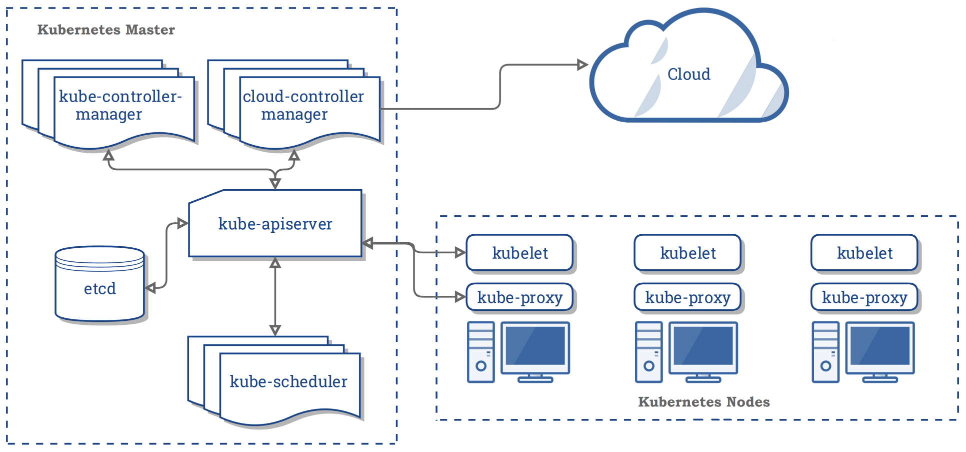


Figure 2.5: Kubernetes Architecture [41]

components and worker components.

2.5.1.1 Master Components

Master components are the control plane of the cluster as they provide core functionalities related to scheduling, configuration backing store, replication and API server. The following components represent the master components:

1. **API Server**: A component that handles Kubernetes API which define how the communication in the cluster happened and called kube-apiserver.
2. **Configuration Store**: A component that provides a consistent and reliable data store for the whole cluster data where etcd [21] key-value store is used.
3. **Scheduler**: A component called kube-scheduler that keeps watching the created pod (group of containers) and select the available node to assign workload for executing.
4. **Controller Manager**: A demon that embeds multiple controllers inside it where it keeps watching the state of the cluster using API server and make the required decision to change the current state to the desired one. Node controller checks and responds when nodes go down, Replication controller maintains the

[21]etcd: `https://etcd.io/`

desired number of pod replicas, Endpoint controller links between the service and pod objects, Service Account controller handles service account creation for new namespaces.

2.5.1.2 Worker Components

Worker components responsible only for running the actual workloads where pods are running. The following components represent the worker components:

1. **Agent**: A component runs in each worker in the cluster called kubelet where it guarantees to run the scheduled pods triggered from the scheduler component. It also register the node with cluster.
2. **Network Proxy**: A component runs in each worker in the cluster called kube-proxy that helps to forward users requests to the desired pods.
3. **Container Runtime**: A component runs in each worker in the cluster where it manages the full lifecycle of containers. Kubernetes does not support only Docker, it also supports containerd [22], cri-o [23] and rktlet [24].

2.5.2 Docker Swarm

Docker Swarm is an open source container orchestration platform licensed under Apache 2.0 that supports running cluster of Docker Engines natively because it is built into Docker Engine which enables developers to use Swarm mode easily. Docker Swarm can be deployed using multiple nodes where the node represents a Docker Engine installed on top of virtual or physical machine. Figure 2.6 shows the Docker Swarm architecture where the following terms are important to understand it:

1. **Manager Node**: A node responsible for handling management functionalities of cluster which includes tasks scheduling, maintaining cluster state, processing API requests.
2. **Worker Node**: A node where the submitted tasks are running into. Manager node decides on which worker node the tasks should be placed in.

[22]containerd: `https://containerd.io/`
[23]cri-o: `https://cri-o.io/`
[24]rktlet: `https://kinvolk.io/`

3. **Service**: A template defines the tasks that need to be executed on the manager or workers nodes. Each task represents a container to run where it contains definition related to image type, number of containers replica and ports in which container will listen to.

4. **Task**: An execution unit where is dispatched to one of the nodes in the cluster in order to run and execute container.

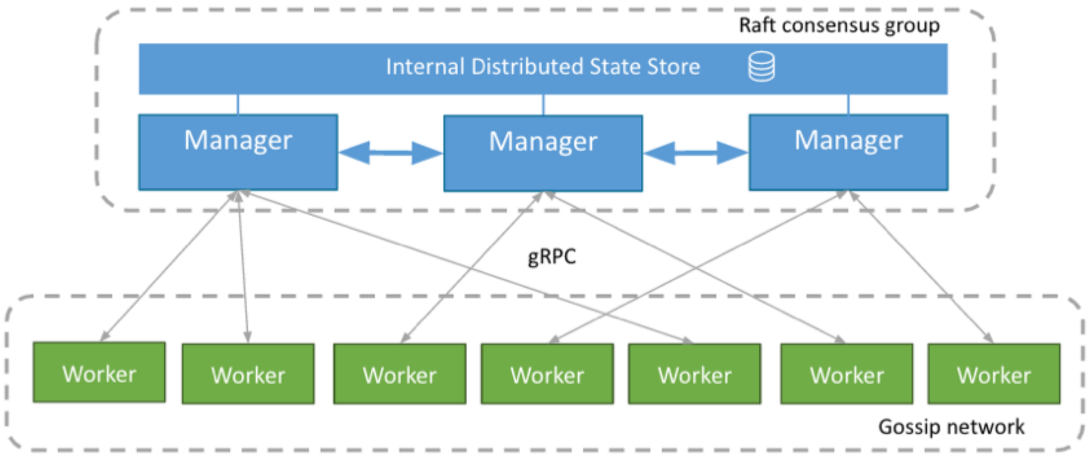


Figure 2.6: Docker Swarm Architecture [9]

2.5.3 Nomad

Nomad is an open source orchestration platform developed by Hashicorp and licensed under Mozilla Public License 2.0. Nomad is not only limited for containerized software but it also can be used with legacy and non-containerized software. Nomad cluster can be deployed using multiple machines where the responsibility of these machines focus on running the actual tasks and managing the whole cluster. Figure 2.7 Shows a high level architecture of Nomad where the following terms are important to understand Nomad:

1. **Client**: A machine where the actual tasks will run. Every client in the cluster will run a Nomad agent that helps the registration with servers. Moreover, it keep watching any submitted task by servers in order execute them.

2. **Server**: A machine responsible for managing the whole cluster. cluster usually contains multiple servers as followers and one leader per region in order to increase the availability and handles fail-over scenarios. Moreover, server responsible for managing submitted jobs, clients, scheduling and task allocation.

3. **Job**: A template specification added by users which declares the tasks need to be run inside Nomad cluster. It contains one or multiple task groups.

4. **Task Group**: A group of tasks which must be run as a whole unit, on the same client and cannot be split.

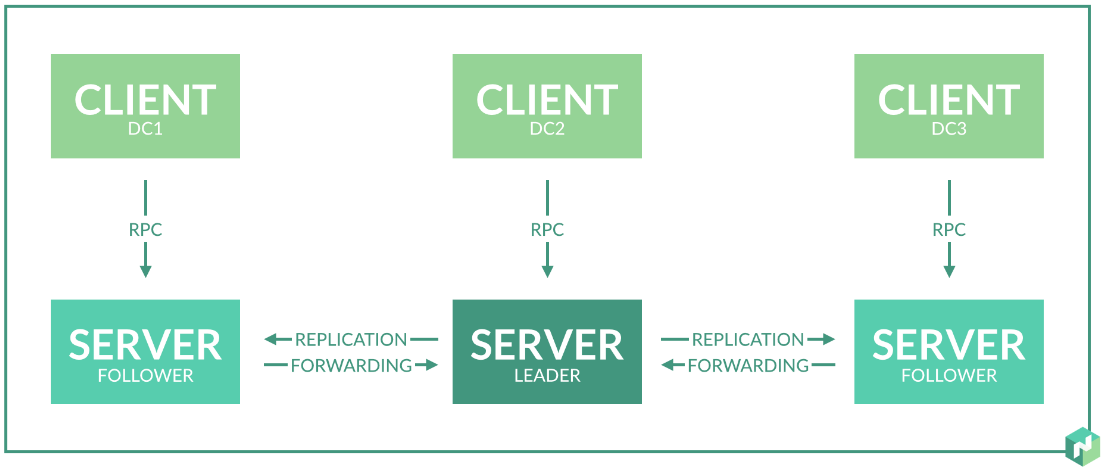


Figure 2.7: Nomad Architecture [49]

2.6 Serverless Computing

Adopting and shifting the software architectures towards microservice and containers lead to emerge a new cloud computing paradigm called Serverless computing for running and deploying software. Serverless computing allows developers to use simple programming model when it comes to create and deploy software to cloud that hides the complexity of the operational tasks. Moreover, Serverless computing pricing model only charges for the execution time instead of resource allocation which reduces the cost of the deployed software. Serverless computing is a commercial buzz word that describes a new programming model for how small code snippets can be run and executed in the cloud without worrying about the underlying infrastructure where the code runs [111]. It does not mean that there are no servers at all, but it gives developers the flexibility to focus on writing their own code and leave the infrastructure management related to provisioning, destroying, scaling and maintenance to the cloud provider. Figure 2.8 shows how the abstraction is increased and concerned about managing infrastructure is decreased while shifting towards Serverless. It also obvious that function is the smallest unit of computing and sometimes Serverless computing called function-as-service (FaaS).

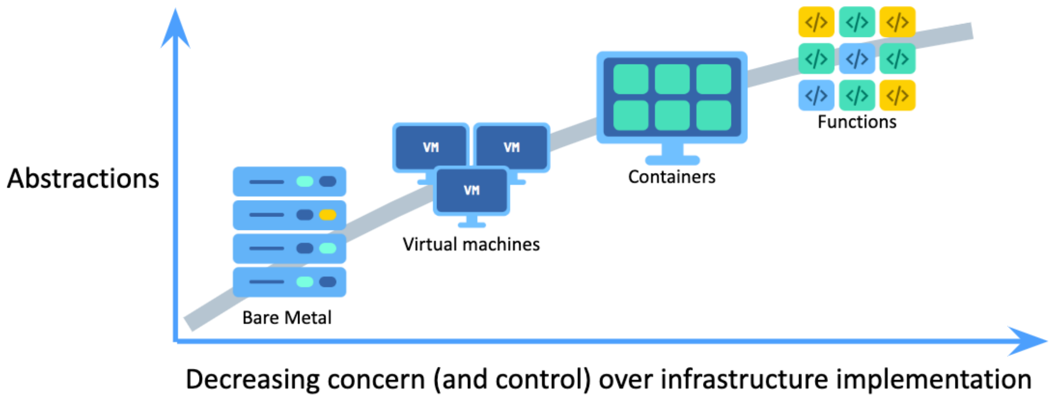

Abstractions
Bare Metal
VM
VM
VM
Virtual machines
Containers
Functions
Decreasing concern (and control) over infrastructure implementation

Figure 2.8: Trends Toward Serverless [73]

Severless computing is a adopted by most of the major cloud providers including Amazon, Google, Microsoft where Amazon is the leading in this field.

2.7 Serverless Frameworks

The most Serverless frameworks which support multiple container orchestrators are OpenFaas, OpenWhisk, IronFunctions and Fn frameworks. In the next subsections, we are going to discuss them in details.

2.7.1 OpenFaas

OpenFaaS [62] is an open source Serverless computing framework under MIT license. It can be installed and deployed to public/private clouds and run on top of multiple container orchestrators including Kubernetes, OpenShift, Docker Swarm, Nomad and can also run on DC/OS [25]. The programming model of OpenFaaS is based on functions [86]. It has an API gateway which is a RESTful component helps to access functions and scale them automatically when get invoked by Alert Manager component which retrieve different metrics from Prometheus. API gateway communicates with underlying providers using provider (i.e., Kubernetes, Nomad, ..etc) to scale functions based on demands. Watchdog is another component acts as HTTP web-server for handling user requests. OpenFaaS has a command line interface which helps to package functions and deploy them as containers. Figure 2.9 shows the OpenFaaS architecture and Figure 2.10 provides a conceptual model of how OpenFaaS components interact with each other. OpenFaaS gives developers the ability to run functions in any programming language they want. Functions are packaged as Docker images which can be deployed and run in top of container orchestrator. Moreover, OpenFaaS provides some ready templates for using different programming languages C#, Go, NodeJS, Python and Ruby and also allows developers to build custom templates if they need to. On the other hand, functions can be invoked using HTTP and other event sources.

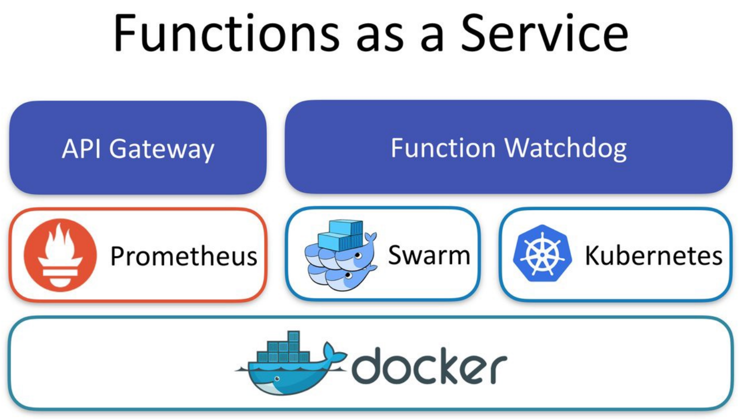


Figure 2.9: OpenFaaS Architecture [62]

[25]DC/OS: `https://dcos.io/`

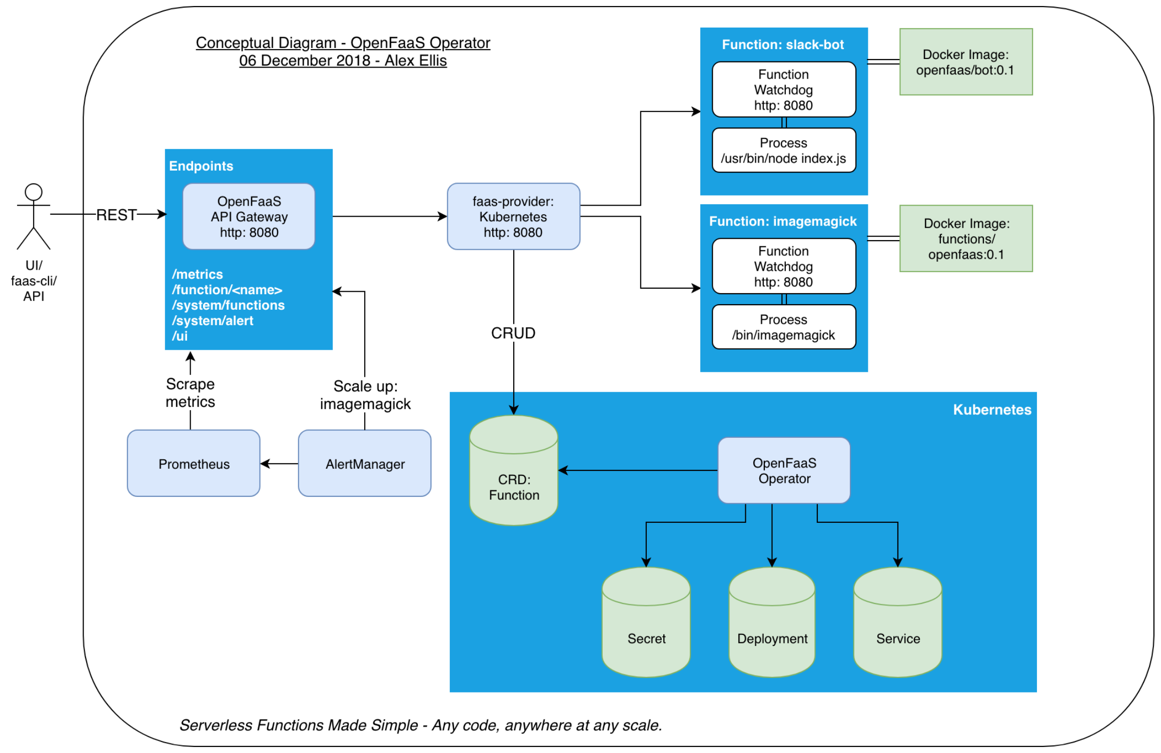

Conceptual Diagram - OpenFaaS Operator
06 December 2018 - Alex Ellis
UI/ faas-cli/ API
REST
Endpoints
OpenFaaS API Gateway http: 8080
/metrics
/function/<name>
/system/functions
/system/alert
/ui
faas-provider: Kubernetes http: 8080
Function: slack-bot
Function Watchdog http: 8080
Process /usr/bin/node index.js
Docker Image: openfaas/bot:0.1
Function: imagemagick
Function Watchdog http: 8080
Process /bin/imagemagick
Docker Image: functions/ openfaas:0.1
CRUD
Scrape metrics
Scale up: imagemagick
Prometheus
AlertManager
Kubernetes
CRD: Function
OpenFaaS Operator
Secret
Deployment
Service
Serverless Functions Made Simple - Any code, anywhere at any scale.

Figure 2.10: OpenFaaS API Gateway Flow [59]

2.7.2 OpenWhisk

Apache OpenWhisk [5] is an open source Serverless computing framework developed by IBM and later incubated by Apache Foundation under the Apache 2.0 license. The programming model of OpenWhisk is based on three primitives: functions, rules and triggers [116] as shown in Figure 2.11. An Action is a stateless function which is invoked as a response to an event and produces result. Trigger is class of events generated from various sources and Rule map trigger to one/multiple actions. OpenWhisk supports multiple container orchestrators which includes: Kubernetes, Openshift and Mesos [26]. It also support multiple programming languages and can be triggered using HTTP, message queue and various events. OpenWhisk has 6 main components specified in Figure 2.12 as the following:

1. **Nginx** [27]: A web server acting as an entry point of the whole system and used as a reverse proxy and forward all requests to the controller component.
2. **Controller Component**. A RESTful API gateway for deployed actions which helps to handle routing actions, authentication, authorisation and expose end-points for CRUD operations on OpenWhisk models.
3. **CouchDB Database** [28]. Store actions created by framework which includes the actual code and their parameters and loades them from the controller once the requester is authenticated. It also saved the results of the executed function.
4. **Invoker component**. The invoker loads the action from database and prepare new Docker container to run action code inside it. It also helps to decide when to invoke new container or re-use existing one.
5. **Apache Kafka** [29]. It acts as message hub between controller and Invoker to allow communications between them.
6. **Consul** [30]. Is a Distributed key-value store, which tracks the state of the OpenWhisk installation [104].

[26]Mesos: `http://mesos.apache.org/`
[27]Nginx: `https://www.nginx.com/`
[28]CouchDB: `https://couchdb.apache.org/`
[29]Apache Kafka: `https://kafka.apache.org/`
[30]Consul: `https://www.consul.io/`

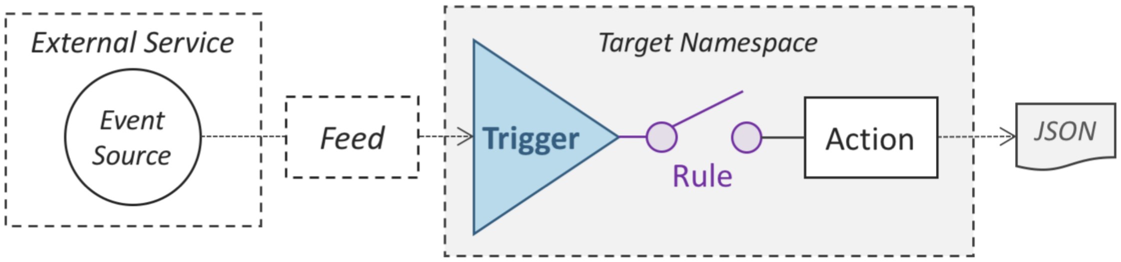


Figure 2.11: OpenWhisk Programming Model [1]

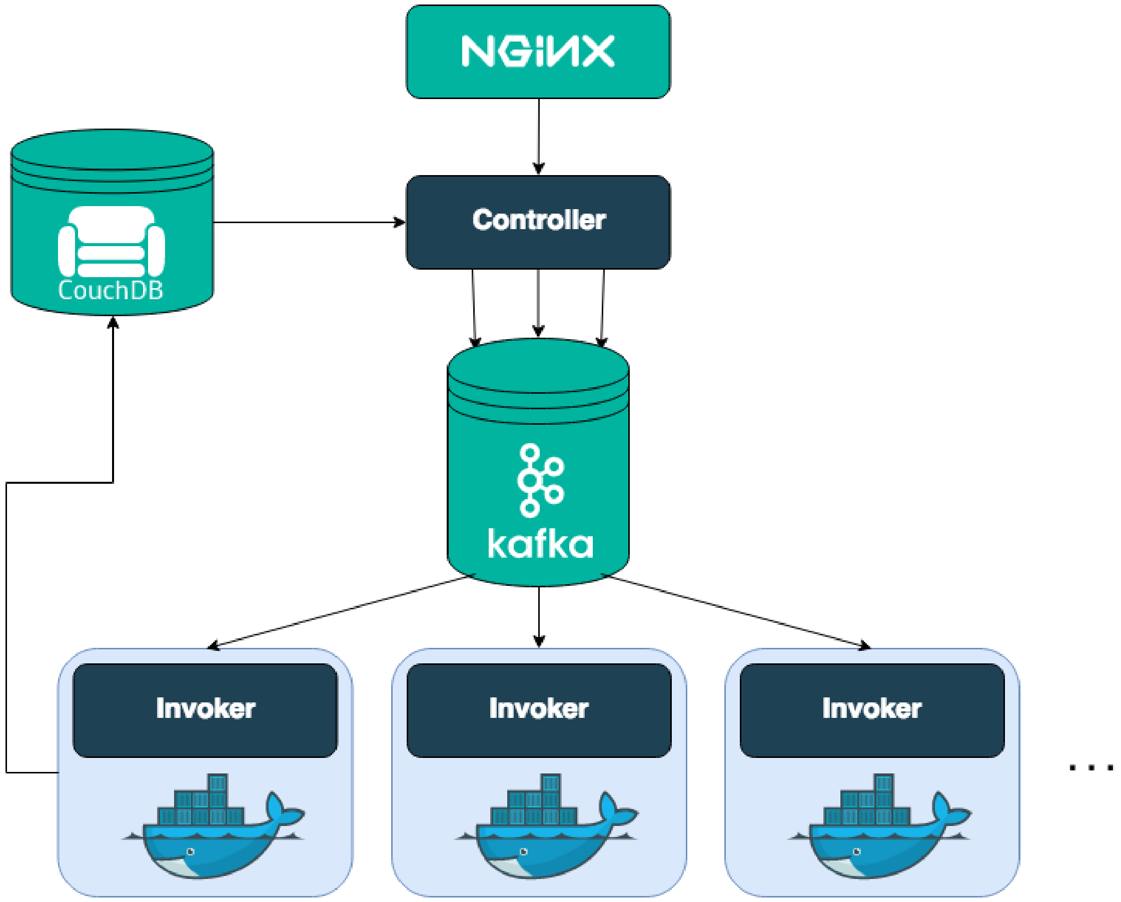


Figure 2.12: OpenWhisk Architecture [1]

2.7.3 IronFunctions

IronFunctions [20] is an open source Serverless computing framework developed by Iron.io [31] licensed under Apache 2.0. The main components of IronFunctions are Fn, Functions, Runner and go-dockerclient [98]. The Fn is the CLI component which helps in functions deployment, building container image, configuration route to server. The Functions component is handling all requests generated from client and forward them to deployed functions based on the routing configuration using CLI. The go-dockerclient [32] component is a client implemented in GO language and used by IronFunctions server side to communicate with Docker remote API. The Runner component contains a collections of interfaces that allows to connect functions and go-dockerclient. The IronFunctions supports two main container orchestrators: Kubernetes and Docker Swarm. Functions can be deployed using multiple programming languages and they can be triggered using synchronous and asynchronous calls. Figure 2.13 shows IronFunctions Architecture In Production.

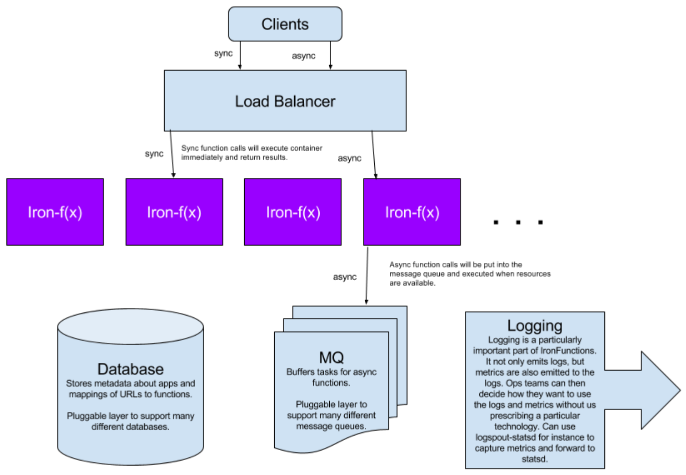

Clients
sync
async
Load Balancer
sync
Sync function calls will execute container immediately and return results.
async
Iron-f(x)
Iron-f(x)
Iron-f(x)
Iron-f(x)
. . .
async
Async function calls will be put into the message queue and executed when resources are available.
Database
Stores metadata about apps and mappings of URLs to functions.
Pluggable layer to support many different databases.
MQ
Buffers tasks for async functions.
Pluggable layer to support many different message queues.
Logging
Logging is a particularly important part of IronFunctions. It not only emits logs, but metrics are also emitted to the logs. Ops teams can then decide how they want to use the logs and metrics without us prescribing a particular technology. Can use logspout-statsd for instance to capture metrics and forward to statsd.

Figure 2.13: IronFunction Architecture In Production [67]

[31] Iron.io: `https://iron.io/`

[32] go-dockerclient: `https://github.com/fsouza/go-dockerclient`

2.7.4 Fn

Fn [77] is an open source Serverless framework developed by Oracle [33] licensed under Apache 2.0. The programming model of Fn is based on 2 primitives: function and events. Function is simply the deployed code wrapped as docker image and events are triggers which execute Fn functions. Figure 2.14 shows how function deployed using Fn framework. Fn contains two main components: CLI and Fn Server. The CLI helps to generate, configure, deploy and invoke functions using RESTful based communication with Fn server. The Fn server handles API gateway, CRUD operations for events and functions, execute functions calls Synchronous/Asynchronous and storing logs. The framework is pluggable and can be integrated with different set of Databases, Queues, Monitoring tools and supports load balancing between multiple node instance of Fn server as specified in Figure 2.15. Moreover, the Fn framework supports multiple programming languages for developing functions and can be deployed to container orchestrators which includes: Kubernetes and Docker Swarm (Not officially support).

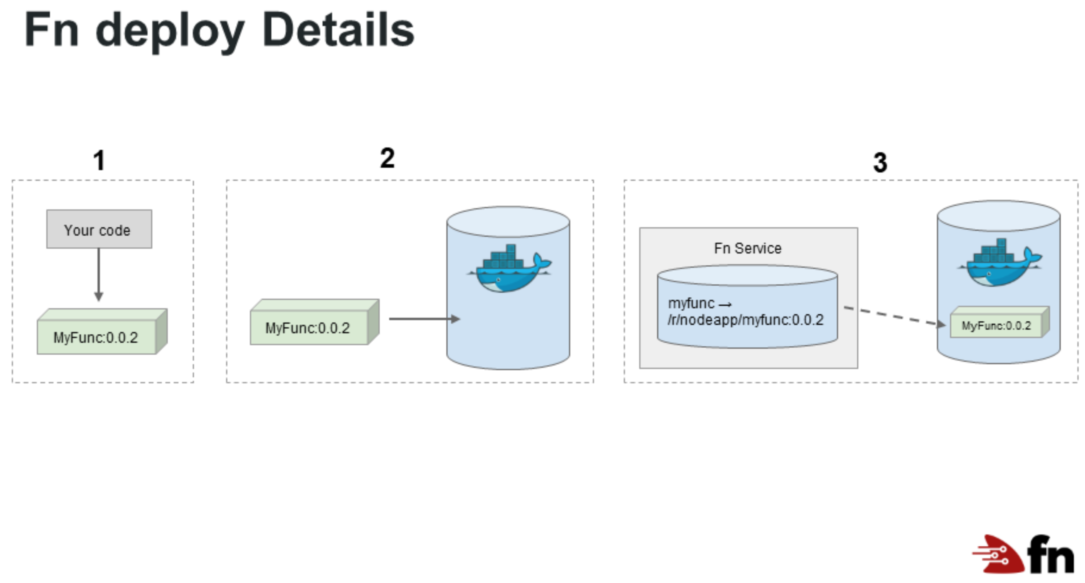


Figure 2.14: Function Deployment in Fn [14]

[33]Oracle: `https://www.oracle.com/index.html`

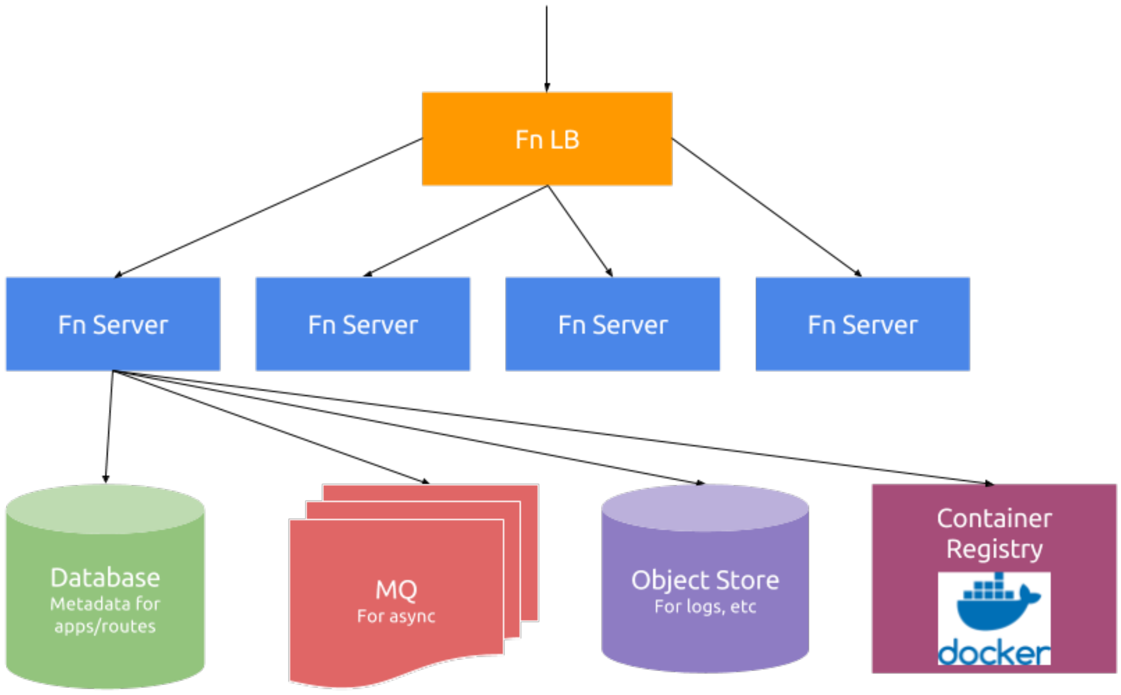


Figure 2.15: Fn Architecture [15]

Chapter 3

Literature Review

This chapter will cover detailed review of related works which studied the evaluation of Serverless functions and the factors that influence their performance. Although, Serverless functions in literature gaining attention from IT practitioners and academics alike [107]. This chapter will address the evaluation of the Serverless functions performance which are deployed on public cloud and on on-premise Infrastructure. According to the literature analysis, the most common metrics used to verify the performance of the Serverless functions are throughput and Response Time. The first section will present the works related to the performance of the Serverless functions that are deployed on public cloud infrastructure and the second section will present the performance of the Serverless functions that are deployed on on-premise Infrastructure using container orchestrators. This research will primarily focus on deploying the Serverless functions on different container orchestrators; in order to measure their impact on performance of Serverless functions. Table 3.1 represents the summary for all related works addressed on this study.

Work	Cloud Environments	Performance Metrics	Container Orchestrator	Use Cases
G. McGrath and P. R. Brenner [113]	AWS Lambda, Azure Functions, IBM Apache OpenWhisk, .NET Prototype Serverless	latency, throughput	No	Node.JS function Windows Containers, Concurrent levels (1 - 15) Cold start tests
W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara [101]	AWS Lambda, Azure Function	response time	No	C# function CPU operations, Cold start tests, Concurrent levels (1 - 100)
D. Jackson and G. Clynch [95]	AWS Lambda, Azure Function	response time	No	Java, Node.js, Python, Go and .NET Core functions Cold start tests
Endreß, T. Heckel, and G. Wirtz [103]	AWS Lambda, Azure Functions	response time	No	Java, Node.js CPU operaions, sequential test cases,
J. Manner and G. Wirt [83]	Simulation and benchmarking model	response time	No	Simulation scenario for the number of created containers
C. Völker [109]	AWS EC2, AWS Lambda	response time	No	Java Spring Application, Node.js function User-based web application Concurrent requests
H. Lee, K. Satyam, and G. Fox [100]	Amazon Lambda, Azure Functions, Google Functions, IBM OpenWhisk	throughput, response time	No	Node.js, Java, C#, and Python functions. Sequential and parallel invocations, CPU, I/O, Network operations,
T. Back and V. Andrikopoulos, [89]	AWS Lambda, Google Cloud Function, Azure Functions, IBM Apache Whisk	response time	No	Node.js, Memory/CPU operations, Sequential requests
K. Figiela, A.Gajek, A.Zima, B.Obrok and M.Malawski [102]	AWS Lambda, Google Cloud Function, Azure Functions and IBM Apache Whisk	response time	No	C wrapped by Node.js function CPU operations
R. Pellegrini, I. Ivkic, and M. Tauber [87]	On Premise	response time	Yes	Node.js function Simple operation (count letters)
S. K. Mohanty, G. Premsankar, and M. D. Francesco [105]	On Premise	response time, success rate	Yes	Go function
A. Palade, A. Kazmi, and S. Clarke [86]	On Premise	response time, throughput, success rate	Ye	Node.js function Concurrent levels (1 - 20)
K. Kritikos and P. Skrzypek [99]	Evaluation Study	N/A	Yes	N/A
S. Shillaker and P. R. Pietzuch [108]	On Premise	throughput	No	Java function moderate to high workloads

Table 3.1: Related Works Summary

3.1 Serverless Computing on Public Cloud

The Serverless term is a new generation of PaaS offered by most of the cloud providers. This new service was spearheaded by AWS Lambda [1] and new services such as Apache OpenWhisk [2], Google Cloud Functions [3], Azure Functions [4] and Iron Functions [5] have emerged as Serverless functions providers [6] where software logic can split into multiple functions and invoked in response to events [113].

G. McGrath and P. R. Brenner [113] studied Serverless functions performance using latency and throughput as evaluation metrics. A Serveless functions prototype was implemented in .NET where it contained the basic components needed to run Serverless functions as it had web service, workers, data storage and message queues. They conducted performance tests on their prototype and other cloud Serveless functions providers AWS Lambda, Azure Functions and IBM Apache OpenWhisk. They conducted an experiment by designing two experiments to compute the performance of the prototype, and then compare its performance with other public Serverless functions. They developed a performance analysis tool to collect the experiment data that is related to the latency and throughput of Serverless functions. As part of the experiment they used a framework called Serverless [7] to help deploying Node.js [8] functions to the different cloud Serveless functions providers and for the prototype as well. They run concurrent tests in order to measure the number of responses received per second and this was implemented by increasing the level of concurrency gradually from 1 to 15 concurrent users. In general, their prototype demonstrated better performance than other frameworks. They also conducted a backoff test in order to measure the latency which mainly targeted the cold start time for Serverless functions. The functions invocation were done during increasing intervals from one to thirty minutes. The prototype latency performance was not bad but Google Cloud Function and AWS Lambda had the best performance. Their study only used one programming language Node.js when they conducted the experiment but using other programming languages could affect the provided result and they only focused on Windows containers. Moreover, they measured latency and throughput for single

[1]AWS Lambda: `https://aws.amazon.com/lambda/`

[2]Apache OpenWhisk: `https://openwhisk.apache.org/`

[3]Google Cloud Functions: `https://cloud.google.com/functions/docs/`

[4]Azure Functions: `https://docs.microsoft.com/en-us/azure/azure-functions/`

[5]Iron Functions: `https://www.iron.io/`

[6]Serverless functions providers: The Serverless functions platforms provided by public cloud

[7]Serverless Framework: `https://serverless.com/`

[8]Node.js: `https://nodejs.org/`

function execution.

W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara [101] investigated the factors which influence microservice performance using Serverless functions in order to give indication about Serverless computing infrastructure differences and enable better software deployments. They studied the performance implications of hosting related to infrastructure elasticity, load balancing, provisioning variation, infrastructure retention, and memory reservation size. The research demonstrated how microservice performance varies depending on four different states of Serverless infrastructure which includes: provider cold, VM cold, container cold, and warm. They used two Serveless functions providers AWS Lambda and Azure Function.

An experiment was setup in order to examine the Serveless functions performance and infrastructure management by developing and deploying function to AWS Lambda and Azure functions and they achieved that using multiple performance tests under different load/stress levels . For Lambda functions, they developed CPU-bound functions that performed random math calculations by defining multiple stress levels for calculations functions. On the other hand, they developed HTTP-Triggered functions using Azure Function written in C# [9]. For infrastructure elasticity they were able to measure performance implications when leveraging elastic Serverless computing infrastructure for microservice hosting by observing how response time was impacted for COLD and WARM service requests. They noticed an extra infrastructure was created to compensate the initialization overhead of COLD service requests which in turn affects any future incoming requests in WARM mode to not use the extra infrastructure created in response to the COLD initialization. They also observed good balanced distribution across hosts and containers of requests at higher calculation stress levels for COLD and WARM service invocations and load distribution was uneven balanced for low stress which causes not to utilize all nodes. They studied the impact of provisioning variation that represents placement of container across VMs on function performance and found out a performance degradation of COLD service up to 4.6x times. Moreover, they studied infrastructure retention to measure how infrastructure was retained and for how long. They identified all the states that could affect their measurement by focus on these four states of Serverless computing infrastructure: provider cold, VM cold, container cold, and warm. After 10 minutes they observed a deprecating in container followed by VMs that causes a performance degradation reaching out 15x after 40 minutes of inactivity. Finally, they studied how memory reservation size could impact the function performance

[9]C#: `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/`

and observed that for COLD service execution time there was a performance increase up to a 4x when increasing function's memory size reservation from 128MB to 1536MB. However, the expected increase was up to 12x since memory was doubled the performance also should be doubled. The increasing performance for WARM service execution time was only a 1.55x improvement and memory reservation beyond 512MB was not helpful to improve WARM service performance. Their study only focused on two public Serveless functions providers AWS and Azure and the result cannot be applied to other providers. Moreover. the function computation implementation in Azure functions was not the same one used in AWS lambda. Finally, they did not specify if they used the same programming language for both providers as they only mentioned the function implemented in Azure.

D. Jackson and G. Clynch [95] focused on their study about how the different programming languages/runtimes could affect performance and subsequent cost of Serverless functions execution by presenting a design and implementation of a new Serverless functions framework for testing to analyze the cost and performance metrics of commercial Serveless functions providers AWS Lambda and Azure Functions. They conducted an experiment for both providers by implementing a completely empty functions in order to abstract out any runtime language effects to measure the impact of different programming languages/runtimes using COLD and WARM start tests. On AWS Lambda, five languages/runtimes were selected Java [10], Node.js, Python [11], Go [12] and .NET Core [13]. Python had the best performance during the WARM start tests but the strange result was GO and Python had better performance in COLD start than WARM start. On the other hand, Azure functions only supports two runtimes Node.js and .NET Core and the last one had the best performance in COLD and WARMS start tests. Using Node.js in AWS Lambda had better performance in COLD start tests than Azure functions. However, .NET Core had better performance than AWS Lambda for both COLD and WARM start tests. They had an unexpected result when COLD start tests had better performance than WARM start tests which requires further investigation. Moreover, the experiments limited only for two Serverless functions providers.

FaaS causes a lot of container starts which requires a lot of cold starts for users. J. Manner, M. Endreß, T. Heckel, and G. Wirtz [103] studied cold start problem

[10]Java: `https://www.java.com/`

[11]Python: `https://www.python.org/`

[12]Go: `https://golang.org/`

[13].NET Core: `https://docs.microsoft.com/en-us/dotnet/core/`

in benchmark and investigated which factors could have impacts on the duration of cold start. They selected AWS Lambda and Microsoft Azure Functions for their benchmark and presented a set of hypotheses as basis for factors that could impact cold starts for functions that included different factors from programming language, platform, function memory size and size of the deployed package.

They conducted an experiment and only considered the following hypotheses: programming Language, deployment package size and memory/cpu settings to be investigated because these were easy to test and could provide stable and reproducible results. They selected two programming languages Java and JS, as Java is a compiled and JS an interpreted language and focused on compute bound operations by selecting a recursive fibonacci function to be deployed into two Serverless functions providers AWS and Azure Functions. They configured the experiment settings to force a cold start closely followed by a warm start on the same container. However, the duration time required for container to startup was not provided by the providers and they had to develop RESTful interactions with the providers that measures start/end time in the client side. On the other hand, they developed a prototype called SeMoDe [69] which created in order to help deploying functions on different Serverless functions provider and automate tests generation. Based on the experiment results, they observed that cold starts were much faster than consecutive warm starts in JS functions deployed to AWS and the explanation because AWS only charges users for functions execution without considering the setup time for virtual machine and containers. They also noticed a gap between compiled and interpreted languages for cold and warm executions using different memory setting where JS execution was much faster than Java. The overhead of cold starts decreased by memory settings and only functions in AWS tested since memory setting cannot be configured in Azure functions. Deployment package size did not give a clear indication if its really impact the cold starts for the deployed functions. Their study evaluated two Serverless functions providers AWS Lambda and Azure Function where container setup and initialization for functions were unknown. The computation functions only measured compute bound operations within two programming languages Java and JS. All the benchmark tests were conducted sequentially and ignore concurrency tests.

J. Manner and G. Wirt [83] presented another work about how application load impacts the Serverless functions on number of running containers. They presented simulation and benchmarking model for deployed Serverless functions which provided useful information about performance and cost at an early stage for development team. The main goal for the research was to be able to conduct a simulation for concurrent running containers for Serverless functions using different configuration settings. They built a generic pipeline for FaaS Benchmarking in order to evaluate the number of concurrent running containers applied against single Serverless function. They considered settings from their previous research [103] such as memory setting and deployment size alongside with load patterns because of their impact on the execution time which helps in assessment for how much concurrent containers were running. The simulation part of the pipeline helps developers to simulate the number of concurrent running containers by adjusting different settings related to memory size and overall execution time that can help them which settings required for better performance and cost. The second part of the pipeline helps in deploying cloud functions by generating workload using different load patterns and the result from invoking these functions will be analyzed and compared against results from the simulation part. Based on the simulation results they concluded that the number of concurrent running containers was impacted using different workloads. Their research used combined pipeline for both simulation and benchmarking but they did not provide and compared results between the simulation part and the results when deploying the Serverless functions using their proptype to see how they are close to each other.

Cloud computing provides multiple paradigms for hosting and deploying software, where performance is a critical factor for software under different workloads. C. Völker, [109] studied the suitability of Serverless functions by measuring the differences between software deployed to IaaS and on Serverless functions providers with respect to cost, performance and response time. The goal was to identify the suitable use cases for Serverless functions approaches. A case study was conducted which aimed to compare for software deployed to IaaS and on Serverless functions approaches by using the same cloud provider AWS. The same test software was built using two programming languages, one of them used Java Spring framework and deployed to AWS EC2 which requires developer to apply all settings required for scaling the software. However, Node.js software was deployed as a Serverless function using AWS Lambda. As a result of his case study using the test software, he concluded that the overall response time for software deployed to EC2 performed better than Serverless functions and the scalability of Serverless was better than EC2. The case study was limited to only one cloud provider AWS and the impact of programming languages was not taken into consideration when the test software was developed using two different programming languages. Moreover, deploying the load test instance was outside the AWS network which could increased the latency and affects the response time for both cases.

H. Lee, K. Satyam, and G. Fox [100] evaluated parallel invocations on Serverless functions under dynamic workloads to measure throughput and performance of Serverless functions. They measured how the performance for CPU, memory and disk intensive tasks differs between sequential and parallel running functions. Moreover, they did a comparison of costs and computation time between Serverless functions and virtual machines. They conducted a set of experiments on four Serverless functions providers Amazon Lambda, Microsoft Azure Functions, Google Functions and IBM OpenWhisk using 4 different programming languages NodeJS, Java, C#, and Python. The purpose was to evaluate Serverless functions and they started that by evaluating the throughput of running functions by thousands of invocations using different triggers (HTTP, Database) on multiple Serverless functions providers. Moreover, they measured how intensive workloads on CPU, Memory, I/O and networks could affect the execution time for sequential and parallel invocations by deploying a test function to all targets providers. On the other hand, they investigated the elasticity of Serverless functions on multiple providers by running a simple test function to check how the overall execution time for invoked function get impacted under heavy concurrent requests. The research only focused on throughput and response time in general and did not consider warm & cold start time for Serverless since this is a very important when studying Serverless. Moreover, the results they shared was not reproducible for the whole tests they did and the usage of the different runtime was not clear enough in all tests.

T. Back and V. Andrikopoulos, [89] evaluated the performance and cost model for the most popular Serverless functions providers AWS Lambda, Google Cloud Function, Azure Functions and IBM Apache Whisk. They conducted a set of tests by developing a microbenchmark called faas-μbenchmark [14] which helped to deploy all functions to the selected providers using different memory settings. To evaluate the performance of different Serverless functions providers they used 3 functions: Fast Fourier Transformation (FFT), Matrix Multiplication (MM) and Sleep function (S). Each function was evaluated using different parameters with different memory settings starting from 128MB to 2048MB. All the implemented functions were developed using NodeJs runtime. Generally, the results of all tests showed that performance was enhanced for increasing memory settings of all the deployed functions. However, the research covered all the tests under one runtime environment based on NodeJs and the performance evaluations was limited to the duration time and did not con-

[14]faas-μbenchmark: `https://github.com/timonback/faas-mubenchmark`

sider other factors such as throughput or latency. Moreover, all the computations were only related to memory/CPU and did not apply any test for I/O or network tasks. Moreover, All the conducted tests did not cover concurrent requests.

Similarly, K. Figiela, A.Gajek, A.Zima, B.Obrok and M.Malawski [102] evaluated the performance of Serverless functions on different providers AWS Lambda, Google Cloud Function, Azure Functions and IBM Apache Whisk. The heterogeneity of Serverless functions was taking into consideration for this research. They used an evaluation benchmarking framework which helped to deploy CPU-intensive functions based on Mersenne Twister and Linpack by using different memory settings on each Serverless functions provider. Their research considered to be unique among previous works conducted a benchmarking on Serverless functions since they addressed a major issue related to heterogeneity of runtime environments on Serverless function providers by building a native binary written in C wrapped by the deployed function written in NodeJS. The result of the research illustrated the heterogeneity between Serverless function providers, especially for how they are handling the resource allocation policies. They also found a relationship between the Serverless function size (allocated memory) and performance. All the conducted tests evaluated only the execution time as performance metric for the deployed functions using different memory settings but did not consider cold time delays for the Serverless functions and all the deployed functions were CPU bound operation only. Moreover, the research did not mention anything related to concurrent invocation to measure its impact on the Serverless function performance.

3.2 Serverless Computing on On-Premise Infrastructure

R. Pellegrini, I. Ivkic, and M. Tauber [87] evaluated the performance of Serverless functions by presenting an architectural model acted as a benchmark tool for Serverless functions. It aimed to give an insight about Serverless environments and identify the factors which may impact the performance of Serverless functions. They built a Serverless functions benchmark framework in order to help evaluation the performance of Serverless functions where generation of the workloads invoked from Java client based called FaaSBench. Another component part of the framework was Proxy Cloud Function (PCF) where its responsible for collecting metrics about the Target Cloud Function (TCF) and the Serverless functions provider. The final one was the Target Cloud Function (TCF) which contains the function logic. They conducted tests using the benchmark tool where OpenFaas Serverless framework was selected for the tests on top of VM with normal specification to evaluate the performance of a simple cloud function (TCF) that counts letters. The header size and transmission time between TCF & PCF in both directions were the only metrics measured during the experiments. The Serverless functions benchmark framework they built for the study was good but they only studied the header size and transmission time to evaluate performance which were insufficient. They did not mention if the workload generated for the tests was sequential or parallel and the implemented function for the TCF was very simple that counts the number of letters. Moreover, only one runtime used NodeJs for PCF and nothing was clear about TCF runtime. They did not take into consideration the network latency between workload generator (FaaS-Bench) and Proxy Cloud Function (PCF) which could affect the result of the tests. All the tests were limited only to one Serverless function framework OpenFaas where they did not mention much information about it.

S. K. Mohanty, G. Premsankar, and M. D. Francesco [105] evaluated the performance of 3 open source Serverless frameworks [15] Fission [16], Kubeless [17] and OpenFaaS [18] by measuring the response time, ratio of successful responses under different workloads and studied the impacts of auto scaling on performance. They also provided detailed features comparison between the selected frameworks. They conducted an experiment where all the tests for the selected Serverless functions frameworks were deployed to Kubernetes [19] orchestrator. First, they disabled the auto scaling feature in all frameworks and apply the tests in fixed replicas (1, 25, 50) subsequently by deploying a simple function written in Go language. They also did the same tests with auto scaling enabled in all frameworks. They found that Kubeless has the most convenient performance in all different test cases. Their work is considered the first evaluation for open source Serverless functions frameworks using Kubernetes orchestrator. However, they only covered Kubernetes container orchestrator and all functions written for test cases were implemented using one programming language Go which could generate different result if other languages were selected. They did not take cold time into consideration when they conducted the test cases.

[15]Serverless functions frameworks: Refers to the Serverless functions framework run in any infrastructure

[16]Fission: `https://fission.io/`

[17]Kubeless: `https://kubeless.io/`

[18]OpenFass: `https://www.openfaas.com/`

[19]Kubernetes: `https://kubernetes.io/`

A. Palade, A. Kazmi, and S. Clarke [86] evaluated the performance of Serverless functions on edge computing environment where 4 open source Serverless functions frameworks were selected Knative [20], Kubeless, Apache OpenWhisk and OpenFaaS. Response time, throughput and success rate of the deployed functions were evaluated under different workloads. They conducted a set of tests on edge computing environment which contained two main layers the IoT Devices Layer where Raspberry PI [21] devices were used as IoT devices and the Edge Computing Layer where all Serverless functions deployed using Kubernetes orchestrator cluster on top of bare metal machines. The generation of the requests initiated by JMeter [22] by different levels of concurrency and the results showed that Apache OpenWhisk had the worst performance among the selected frameworks for all the metrics and Kubless can scale better than other frameworks. Their work was considered the first one which evaluated open source Serverless functions frameworks in the field of the edge computing environment. However, all the test cases were conducted using simple function written in NodeJs and did not consider operations like CPU, Memory, I/O or network which requires further investigations. Moreover, the selected Serverless functions frameworks were deployed using only Kubernetes orchestrator and using other orchestrators (Docker Swarm) could have different results. The results of the research cannot be reproducible.

K. Kritikos and P. Skrzypek [99] evaluated seven open source Serverless frameworks by providing a feature comparison analysis based on predefined criteria related to the software lifecycle. The selection of Serverless frameworks was based on abstraction frameworks where complexity of multiple Serverless frameworks is abstracted away from developers. Moreover, provisioning frameworks were considered where they can be operated as standalone Serverless frameworks which can be deployed on top of container orchestrator (Kubernetes) over existing cloud infrastructures. They studied Serverless frameworks using feature comparison analysis related to software lifecycle. Selecting the best performer candidate between the Serverless frameworks using feature comparison was not enough since they did not do real comparison related to how each framework behaves under different workloads to measure their performance.

[20]Knative: `https://knative.dev/`

[21]Raspberry PI: `https://www.raspberrypi.org/`

[22]JMeter: `https://jmeter.apache.org`

S. Shillaker and P. R. Pietzuch [108] presented at the beginning of their research a plan to build a new multi-tenant Serverless language runtime in order to reduce overhead and obstructing of resource sharing produced by Serverless functions when using in its own Docker [23] container. They claimed that the new approach will reduce latency, improve resource efficiency and provides a smart approach for scheduling. In order to prove the existence of these limitations, they conducted an experiment to study the performance of Openwhisk framework. All the tests were conducted in functions written in Java under moderate to high workloads where they focused on equivalent throughput across different system metrics which includes latency, CPU cycles and Memory using different numbers of function to study the performance of Openwhisk. Moreover, they also studied the behaviour of Openwhisk when the rate of submitted requests bypass the container limit for handling requests where response rate decreased. They explained the results because of how Openwhisk handle scheduling and the isolation imposed by containers. The investigation they did was limited only to one Serverless framework Openwhisk and did not consider other frameworks which could generate different results. All the functions were deployed as standalone containers without using any help from container orchestrator. On the other hand, they did not mention the type of operations executed by the deployed functions and all functions implemented using Java programming language which depends on JVM.

3.3 Discussion and Conclusion

In the previous sections, we presented all works that studied performance of Serverless functions deployed using public cloud providers and other open source frameworks deployed to on-premise infrastructure. There were plenty of related work that evaluated the performance of Serverless functions on public cloud using different Serverless functions providers like AWS Lambda, Azure Functions, Google Functions and IBM OpenWhisk. Some of them only evaluated the performance of Serverless functions using limited cloud providers while others did the evaluation in all available providers. Most of the works conducted all the tests using one programming language/runtime while few of them using multiple languages/runtimes supported by the current cloud service providers. On the other hand, most of the work used common metrics for performance evaluation by conducting their experiments depending on throughput and response time and some few works considered other metrics like latency. All the computations were only related to Memory, CPU for most of the works but none

[23]Docker: `https://www.docker.com/`

of them handle all computations types which includes I/O and networks alongside Memory and CPU. Every public Serverless functions provider has its own implementation, scaling policy, scheduling and cost models for handling requests which helps to abstract a lot of headache for users like how each function is executed inside the provider infrastructure. Usually each function in public providers run inside a container which requires a lot of cold starts for users. Few of the above works in public providers studied the impact of cold and warm starts to the performance. The majority of the work evaluated the performance of Serverless function using concurrent requests and some other works consider both concurrent and sequential requests while few of them did not mention the type of loads being generated. Some of works presented a detailed information on how to reproduce the results of the experiments they conducted using benchmark tools they developed for that purpose while others did not mention anything about that.

The number of works that evaluated the performance of the open source Serverless frameworks deployed to on premise infrastructure were limited. Most of the works used the same performance criteria response time, throughput and others used success rate. Moreover, some of them deployed their functions using container orchestrator like Kubernetes since most of the open source frameworks can be run as Docker container which can get extra credits for load balancing, networking and container management features for free [108]. However, none of the related works studied if different container orchestrators could impact the performance of deployed Serverless functions. On the other hand, None of them used multiple programming languages/runtimes when they conducted the experiments which could impact the results. The implementation of Serverless functions of all previous works on on-premise infrastructure contained simple logic and did not handle complex operations related to Memory, CPU, I/O or networks. The cold and warm starts are an important factors that was not considered in previous works either. Moreover, none of the related works that studied performance of Serverless using open source frameworks provided full details about reproducing the results so that we can re-use or validate them in our research.

Serverless functions can be run as standalone Docker containers which opens the door for researchers to study Serverless functions using container orchestrators like Kubernetes to help in load balancing, routing and container lifecycle management. Investigating if different container orchestrators can impact the Serverless function performance is an interesting research topic this thesis tries to address. Kubernetes, Docker Swarm and Nomad are examples of container orchestrators can be used to measure Serverless functions performance by focusing on throughput, Response time and Success Rate under different function computations Memory/CPU, I/O and networks. Moreover, multiple functions will be deployed using different programming languages/ runtimes NodeJs, Python, Go and Java with concurrent and sequential requests. The impact of cold and warm start time will be considered as part of this research. More details are going to be explained in methodology chapter.

Chapter 4

Research Methodology

The main goal of this chapter is to provide a clear understanding about the research methodology used to evaluate the performance of the Serverless framework deployed on top of different container orchestrators and provides details about data collection, analysis, design and setup of of the research method. A. Palade, A. Kazmi, and S. Clarke [86], S. K. Mohanty, G. Premsankar, and M. D. Francesco [105] evaluated performance of different open source Serverless frameworks on top of Kubernetes orchestrator in terms of response time, throughout and success which is the same approach we followed in this study by using the same metrics. In order to investigate the impact of different container orchestrators on Serverless, we conducted an experiment to check how the performance will be affected under different container orchestrators for deployed Serverless functions. Section 4.1 provides details about the methodology used for performance evaluation. Section 4.2 provides details about data collection. Section 4.3 discusses the data analysis. Section 4.4 illustrates the evaluation for Serverless frameworks that we did for 9 frameworks. Section 4.5 discusses the performance metrics and factors for Serverless functions. Section 4.6 provides insights about the required procedures for performance evaluation of Serverless functions. Section 4.7 discusses the architecture used to evaluate the performance. Finally, Section 4.8 provides details about experiment design and full details about experiment tool

4.1 Research Approach

In this thesis, a quantitative research approach was selected to evaluate the Serverless performance on top of different container orchestrators. Experiment was the best approach to choose as it provided statistical data to find a relationship between variables (independent and dependent) to test our hypotheses. Our research was extremely fit with experiment approach since it investigated the relationship between deployed Serverless functions and container orchestrators in terms of performance using different settings and controllable environment. Moreover, the investigation focuses on performance metrics particularly response time, throughout and success rate and these are continuous measurement. We build an experiment with a systematic steps that help in replicating and reproducing the results easily depending on custom experiment tool built for this purpose.

4.2 Data Collection

As prerequisite of the experiment, an evaluation of Serverless frameworks which supports running on different container orchestrators was conducted as illustrated in section 4.4 to choose the suitable framework for this research since the base Serverelss framework (OpenFaaS) to work with was the initial step to move forward to start our investigation. Collecting the data of submitted requests was required to simulate generation of http requests with different settings in order to cover all the test cases that measure the performance metrics where OpenFaaS was deployed to the selected container orchestrators Kubernetes, Docker Swarm and Nomad. As the number of generated test cases are quite large, handling them manually even with using one of the load testing tool required a lot of effort and time and here it came the idea of building an automation tool to deal with that where *faas-exp* [1] was developed for that purpose. *faas-exp* was the entry point of generating the http requests and gather all the raw data to be used later on for analysis where more details is illustrated at Section 4.8.6.

4.3 Data Analysis

As we discussed previously, we are interested in studying the performance of Serverless using different container orchestrators based on the selected metrics defined. The *faas-exp* played a crucial role on conducting the experiment where it focused on generation test cases, aggregating raw data, analysing them and generating visualizing figures. *Median* is used to do the comparison between the three container orchestrators for all defined test cases based on the performance metrics selected for the experiment. The reason for selecting the *Median* was to avoid the outliers values that

[1] faas-exp: `https://github.com/mabuaisha/faas-exp`

could affect the generated results and in order to accurately reflect the typical value specially for response time, throughput as contained some outliers values. Moreover, IQR (Interquartile range) was also used in order to detect the outliers values and remove them before aggregate the results. On the other hand, we introduced quite amount of factors in this research to study the impact the of container orchestrators on the Serverless function performance and to make sure that all the results we obtained from this research not explainable by chance we applied a statistical analysis using *Wilcoxon* paired tests in order to validate that all the results are statistically significance by testing the *p-value*.

4.4 Serverless Frameworks Evaluation

In this section, most of Serverless frameworks that can be run as container and deployed to orchestrators are evaluated following the same approach did before by [86] [105] [104] but with focus more on frameworks that support orchestrators such as Kubernetes, Nomad, Docker Swarm and Mesos. The evaluation contains new frameworks never evaluated before like Fn, Nuclio and Kyama. It will help to select the most relevant Serverless framework with multiple orchestrators support to conduct this research. This research evaluated 9 Serverless frameworks where frameworks illustrated in Table 4.1 support single container orchestrators and frameworks illustrated Table in 4.2 support multiple container orchestrators. As our research focuses to study these type of Serverelss frameworks, its important to have free access and full control to the framework without any limitations. It is also essential for the Serverless framework to support container orchestrators because the main goal of the research is to investigate performance of Serverless functions deployed to different container orchestrators. Moreover, In order to answer our research questions it is required for Serverless frameworks to support multiple programming languages and to be invoked using HTTP event source. Logging and monitoring all requests against Serverelss frameworks are important to capture important metrics for analysing purposes. The ability to scale functions based on demand is also impacts performance which usually should be supported by the framework. Support of CLI allows easy deployment and invocation of functions when conducting load testing. The evaluation features selected for Serverless frameworks are specified as the following:

1. **Open source license**. All the selected frameworks are open source which give developers the freedom to access and use them without restrictions on any infrastructure and avoid vendor lock-in.

2. **Programming Languages Support**. Multiple programming languages support is essential for Serverless framework which give developers the flexibility to use their own language. Moreover, since Serverless framework can be run as container it should be easy to build and develop any function using any programming languages by building custom docker image.

3. **Container Orchestration Support**. Most of the Serverless frameworks support running using container orchestrators which helps in deployment, operation and scaling function if necessary.

4. **Function Triggers**. Functions invocations support various sources and can be synchronous invocation (HTTP), Scheduler invocation (cron) or asynchronous event invocation (message queue).

5. **Monitoring Support**. The ability to measure the performance of Servelress functions is crucial and the selected framework should have the ability to be integrated and configured easily with other monitoring tools (Prometheus [2], Grafana [3]).

6. **Auto Scaling**. Serving requests based on demand is important and the framework should have the support to scale in/out to satisfy user needs.

7. **CLI Support**. Interaction with Serverless functions using command line interface is important for configuration and management. It also helps deploying and invoking function and integrate framework easily with third party tools.

8. **Github Community**. Github [4] Developer community is an important factor for evaluation open source software to check the software maturity, stability, development community and how much Serverless framework is popular.

All the 4 Serverless frameworks presented in Table 4.2 support multiple container orchestrators. In order to select the best candidate to use for our research, we tried to install and configure each framework in a testing environment by following their documentation website and deploy simple hello world function in top of different container orchestrators. We found that OpenFaaS framework has clear documentation and steps for deployment functions to different container orchestrators: Kubernetes, Docker Swarm and Nomad and also it supports other container orchestrators that

[2]Prometheus: `https://prometheus.io/`
[3]Grafana: `https://grafana.com/`
[4]Github: `https://github.com/`

Feature	Fission	Nuclio	Kubless	Kyma	Knative
Open source license	Apache License 2.0 [12]	Apache License 2.0 [56]	Apache License 2.0 [39]	Apache License 2.0 [48]	Apache License 2.0 [32]
Programming Languages	NodeJS, Python, Ruby, Go, PHP, Bash, Any custom container executable [2]	.NET Core, Go, Java, NodeJS, Python, Shell [52]	NodeJS, Python, Ruby, Go, Java, .NET Core(C#), Ballerina, Any custom container [37]	Kubeless, Knative are parts of Kyma (Support multiple languages) [43]	.NET Core(C#), Go, Java, Kotlin, NodeJS, PHP, Python, Ruby, Scala, Shell [30]
Container Orchestration Support	Kubernetes [13]	Kubernetes [54]	Kubernetes [42]	Kubernetes [46]	Kubernetes [29]
Function Triggers	http, schedule, message queue and other events [76]	http, schedule, message queue and other events [55]	http, schedule, message queue [38]	http, schedule, message queue [47]	http, schedule, message queue and other events [31]
Monitoring Support	Prometheus [75]	Prometheus [53]	Prometheus, Grafana [36]	Prometheus, Alertmanager, Grafana [45]	Elasticsearch, Kibana, Stackdriver, custom logging [28]
Auto Scaling	Yes [10]	Yes [50]	Yes [34]	Yes [44]	Yes [26]
CLI Support	fission [11]	nuctl [51]	kubless [35]	kyma-cli [81]	kn [27]
Github Contributors	90 [13]	49 [84]	82 [40]	91 [82]	152 [33]
Github Stars	4.8k [13]	3k [84]	5.2k [40]	787 [82]	2.5k [33]
Github forks	432 [13]	290 [84]	531 [40]	221 [82]	495 [33]

Table 4.1: Serverless Reviews Frameworks with Single container orchestrators

Feature	OpenFaas	OpenWhisk	IronFunctions	Fn
Open source license	MIT License [63]	Apache License 2.0 [6]	Apache License 2.0 [21]	Apache License 2.0 [17]
Programming Languages	Go, Python, NodeJS, .NET Core (C#), Ruby, Java, PHP7, Any custom container [8]	NodeJS, Go, Python, Java, PHP, Ruby, Swift, .NET Core, Any custom container [3]	Go, Java, NodeJS, PHP, Python, Ruby, Rust, Any custom container [22], [23]	Go, Java, NodeJS, PHP, Python, Ruby, Rust, Anu custom containers [77], [78]
Container Orchestration Support	Kubernetes, Docker Swarm, Nomad, OpenShift, DC/OS [19], [58]	Kubernetes, Mesos, OpenShift [65], [66]	Kubernetes, Docker Swarm [24]	Kubernetes, Docker Swarm (no offical support) [15], [79]
Function Triggers	http, schedule, message queue, other event sources [61]	http, message queue, other event sources [4]	http, message queue [94]	http [16]
Monitoring Support	Prometheus, Grafana [70]	StatsD Grafana [7]	StatsD [25]	Prometheus, Grafana [18]
Auto Scaling	Yes [57]	Yes [64]	Yes [94]	Yes [77]
CLI Support	faas-cli [85]	wsk [72]	fn [94]	fn [77]
Github Contributors	128 [62]	168 [5]	32 [20]	85 [77]
Github Stars	16.1k [62]	4.4k [5]	2.7k [20]	4.3k [77]
Github forks	1.3k [62]	848 [5]	205 [20]	319 [77]

Table 4.2: Serverless Frameworks Reviews with Multiple container orchestrators

can be deployed to. Moreover, it is more popular than others and easy to use and integrate with. Aforementioned, The OpenFaaS is going to be used as a reference framework in this research using multiple container orchestrators to find if container orchestrator can affect the performance of deployed Serverless functions.

4.5 Performance Metrics and Factors

This research focuses on measuring how the performance of deployed Serverless functions under different container orchestrators can be varied using different set of factors. Performance is a crucial aspect for Serverless functions when it comes to serve user requests for high workloads. The ability of Serverless framework to scale in/out based on different workloads helps to reduce overheads on deployed software, minimize response time and maximize throughout. This will be helpful especially for Microservice architecture where hundreds of functions deployed to handle incoming traffics. The following subsections focus on the following:

1. **Performance Metrics**: The performance metrics which are going to be studied in this research are response time, throughput and success rate.
2. **Performance Factors**: There are some factors that can impact the performance of Serverless which includes: computation types: Memory, CPU, I/O and networks bound operations, using different programming languages, concurrent and sequential requests and impact of cold and warm start time.

4.5.1 Performance Metrics

Serverless is highly tied with Microservice architecture where the whole software can be split into multiple functions and each function responsible for separate functionality. Serverless function can be packaged and deployed as container on top of container orchestrator which is essential to measure and evaluate its performance using different metrics and since there are some Serverless framework such as OpenFaaS supports running under different container orchestrators then the performance of deployed functions can be varied from container orchestrator to another. This research measure the performance of Serverless functions deployed under different container orchestrators as the following:

1. **Response Time**: The total time required to complete user requests from the moment of request generation until the response is received where it measures the performance of an individual request.

2. **Throughput**: The number of completed requests per second where it measures the overall performance of the system.

3. **Success Rate**: The ratio of successful requests to the total number of generated requests.

4.5.2 Performance Factors

The performance of Serverless function varies based on different settings and circumstances. In this research, the base Serverless framework selected is OpenFaaS which supports running using different container orchestrators and there are some settings/factors that need to be studied when deploy Serverless functions on these container orchestrators. This research focuses on the following criteria to measure the impact of container orchestrators on deployed Serverless function based on the performance metrics define before:

1. **Operation Type**: Function can be implemented for executing different type of operations that can be related to CPU/Memory-bound operation (matrix multiplications), I/O-bound operation (Write files) or Network-bound operation (Download files)

2. **Programming Languages/Runtimes**: OpenFaaS supports wide range of programming languages which allows to deploy functions using different languages where the main selected languages/runtimes for this research are Python, Java, Go and NodeJS.

3. **Generated Workload**: The generated workloads that trigger functions can be sequential or parallel. These two types are going to be considered in this research.

4. **Warm and Cold Start Times**: These two factors have an impact on response time/throughput of functions when invoke them during the warm time (container ready to accept requests) and on cold start (function not ready yet and need to be initialized).

4.6 Performance Evaluation Procedures of Deployed Serverless Functions

In order to evaluate the performance of Serverless function deployed on different container orchestrators, it is required to have infrastructure to use for installing, configuring container orchestrator, deploying Serverless function and performance testing. All these steps can help to reproduce results and conduct performance evaluation easily. The goal is to design an evaluation flow focuses mainly on two parts. First, to provision infrastructure, configuring and installing container orchestrator and deploy Serverless functions. Secondly, prepare a performance evaluation by running load tests to measure the desired metrics of this research and analyse the results. The evaluation flow starts with the following steps:

1. **Cloud Infrastructure Provisioning**: Infrastructure refers to the hosting environment which supports the computing requirements resources to run functions and provisioning resources and can be automated using RESTful APIs.

2. **Application Layer Configuration**: Once the infrastructure is ready, different orchestrators can be easily installed and configured on top of the provisioned infrastructure. Moreover, OpenFaaS will also be ready to be installed and configured.

3. **Function Deployment**: Deploy function to target environment (container orchestrator).

4. **Performance Testing**: This part will help to automate generation of requests under different settings.

5. **Data Analysis**: Analyze all the generated results that help to draw a conclusion to explain the relationship between container orchestrators and Serverless function.

4.7 Evaluation Architecture

Previously, we presented a general overview about the evaluation steps used in this research as a baseline of our investigation. This section will provide more details about the evaluation architecture which includes the internal components of each phase and how they interact with each other. Figure 4.1 presents detailed overview about the evaluation architecture where two main layers are taking into consideration: Infrastructure and application layer which are explained in details in the next two subsections. Moreover, Figure 4.2 presents the deployment diagram of all components.

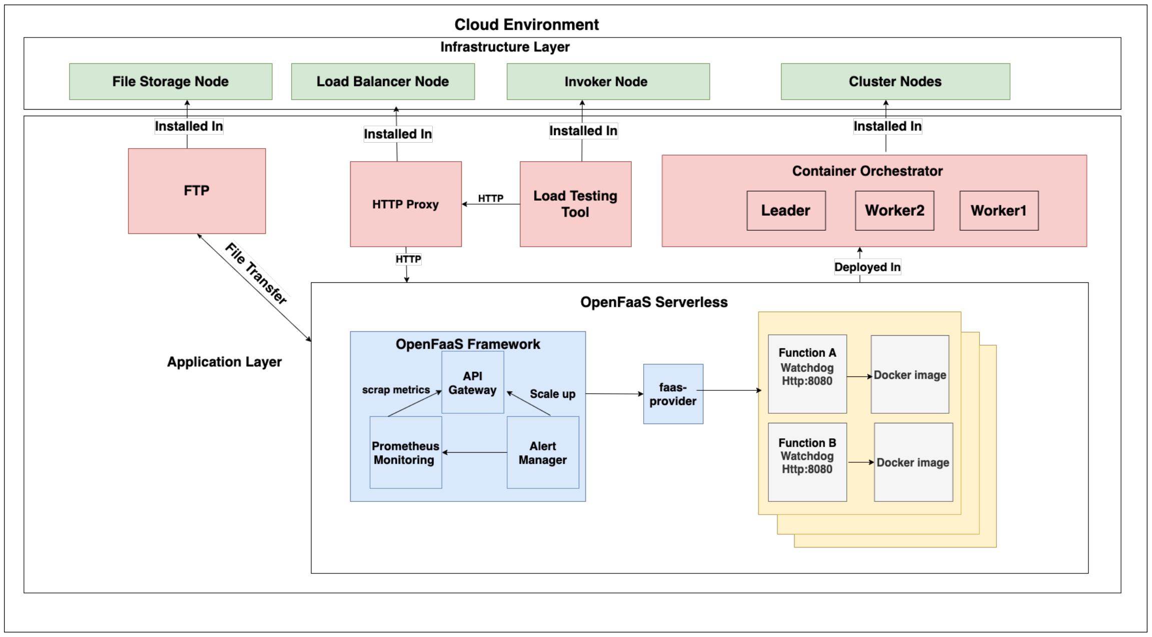


Figure 4.1: Evaluation Architecture

4.7.1 Infrastructure Layer

This layer is an essential part of our architecture where it contains four type of nodes: Cluster, Invoker, Load Balancer and File Server. Node refers to virtual machine responsible for hosting software, functions. All these nodes are provisioned in using AWS cloud infrastructure where nodes can be easily created and destroyed using API enabled by the cloud infrastructure.

1. **Cluster Nodes**: Core nodes where contain all the required software and applications to run Serverless functions in top of container orchestrators. The cluster nodes will contain container orchestrator, OpenFaaS Serverless framework and functions. The size of the cluster for each container orchestrators: Kubernetes, Docker Swarm and Nomad is going to be 3 nodes share the same

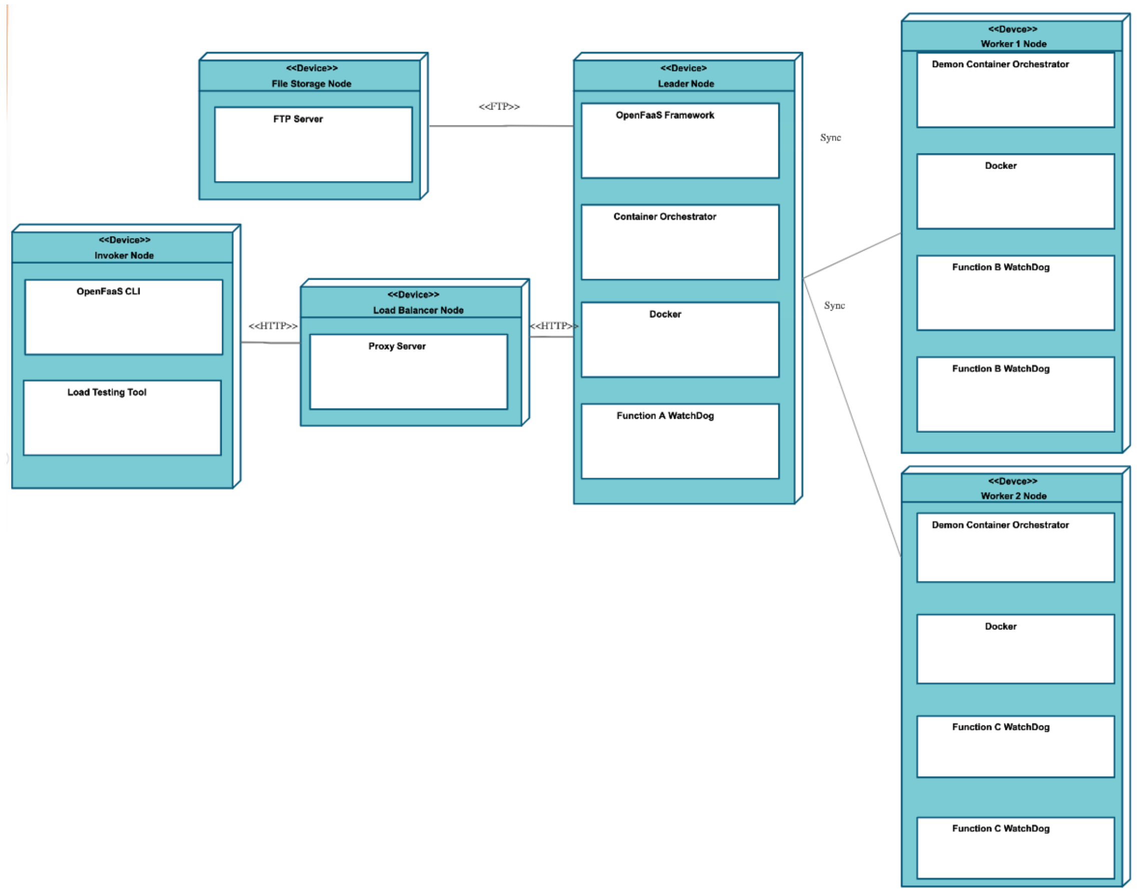


Figure 4.2: Evaluation Deployment Architecture

specification: CPU/Cores, RAM and Volume (Storage) which is going to be explained in Environment Setup Section.

2. **Invoker Node**: This node used for management and generating load testing on the cluster nodes and it resides on the same network of Cluster nodes and File Server to mitigate the network latency. The baseline for all tests will be triggered from this node which contains CLI of OpenFaaS to deploy function in Cluster nodes.

3. **File Storage Node**: This node represents a storage layer of all files requested from the Serverelss function deployed to container orchestrator. As we are going to measure the Network-bound operation in this research, this node is essential to measure this aspect. This node is reside on the same network of other nodes in order to mitigate the network latency from transferring files.

4. **Load Balancer Node**: This node contains the proxy server that act as middleware for handling requests/responses between client and backend servers (cluster workers).

4.7.2 Application Layer

This layer represents two parts. Firstly, the execution environment which is container orchestrator where all functions are deployed and running to serve user requests. Secondly the utility environment which represents the *faas-exp* tool installed inside invoker node and FTP server which serve files requested by Serverless functions.

1. **Container Orchestrator Cluster**: This represents the execution environment of OpenFaaS Serverless framework and all deployed functions. Configuring container orchestrator cluster starts with primary node which responsible for management, scheduling and leading the whole cluster. The primary node has name based on the type of container orchestrator where it is called Master node in Kubernetes, Manager in Docker Swarm and Server in Nomad. Moreover, The secondary node is called worker where it runs the actual workloads and sync with the primary node. Most of the container orchestrator share the same term "Worker" except Nomad "Client". In Figure 4.1 we agree to call the primary node as "Leader" for simplicity and the rest of the node as "Worker". As mentioned before, the size of our cluster is 3 node, 1 leader node and 2 workers.

2. **OpenFaaS Framework**: The OpenFaaS framework itself will be deployed in top of container orchestrator. All its components including: API Gateway, Alert Manager and Prometheus will be deployed as containers. OpenFaaS communicates with container orchestrator using a faas-provider for managing Serverless functions inside the cluster.

3. **Load Testing**: The simulation of http requests will be handled and generated using *faas-exp* tool which is installed on the invoker node. It will communicate with functions deployed in container orchestrator to measure the performance metrics specified before.

4. **FTP Server**: This represents the File Server which is installed inside the File Storage Node in order to serve all files transferred from the Serverelss functions using FTP protocol.

5. **Proxy Server**: This node represents the entry point for all requests generated by the load testing tool where it balances the requests among the cluster workers using round robin algorithm where it serve responses to clients on behave of backend servers (cluster workers).

4.8 Experiment Design

The goal of this experiment is to investigate and find a relationship between the deployed Serverless function and the container orchestrator in terms of performance under different circumstances. In order to accomplish that and before start running the experiment, we need to define the context and boundaries of the experiment which includes the set of Hypotheses need to be investigated, performance metrics need to be measured, specifying dependent and independent variables, the flow of the experiment which covers all the scenarios and finally the environment where the experiment will take place.

4.8.1 Experiment Hypotheses

In order to be able to answer the research questions, the following alternative hypotheses are defined to help answering them:

1. H_1: There is a relationship between response time of deployed Serverless function and container orchestrators.

2. H_2: There is a relationship between throughput of deployed Serverless function and container orchestrators.

3. H_3: There is a relationship between response success rate of deployed Serverless function and container orchestrators.

4. H_4: There is a relationship between generated workload requests (sequential/-parallel) against the response time of Serverless function and container orchestrators.

5. H_5: There is a relationship between generated workload requests (sequential/-parallel) against the throughput of Serverless function and container orchestrators.

6. H_6: There is a relationship between generated workload requests (sequential/-parallel) against the success rate of Serverless function and container orchestrators.

7. H_7: Using different operation types of the Serverless function will affect the response time of function deployed to different container orchestrators.

8. H_8: Using different operation types of the Serverless function will affect the throughput of function deployed to different container orchestrators.

9. H_9: Using different operation types of the Serverless function will affect the success rate of function deployed to different container orchestrators.

10. H_{10}: Using different programming languages/runtimes will affect the response time of function deployed to different container orchestrators.

11. H_{11}: Using different programming languages/runtimes will affect the throughput of function deployed to different container orchestrators.

12. H_{12}: Using different programming languages/runtimes will affect the success rate of function deployed to different container orchestrators.

13. H_{13}: There is a relationship between cold/warm Start times of the Serverless function response time and container orchestrators.

14. H_{14}: There is a relationship between cold/warm Start times of the Serverless function throughput and container orchestrators.

On the other hand, the Null hypotheses H_0 of the defined metrics are to reject the relationship between the defined factors and the container orchestrators.

4.8.2 Experiment Metrics

Performance is what going to be investigated in this experiment which mainly focuses on the metrics we illustrated previously in details which are response time, throughout and success rate. These metrics will be measured when deploy OpenFaaS to the predefined container orchestrators.

4.8.3 Experiment Variables

Conducting experiments help to establish relationships between cause and effect. Cause and effect provide explanation, answer why things happen and help in predication what will happen when apply something. Experiment is designed to measure if something changes will cause something else to change in repeatable fashion. Experiment variables are the things which are changing where they represent factors or conditions. An experiment has three types of variables: independent, dependent, and controlled. For this experiment which focuses on measuring the performance of Serverless function deployed to different container orchestrators, we are going to specify all variable types related to our research.

4.8.3.1 Independent Variables

Independent variables refers to variables which can change during the experiment. The following variables represent the independent variables used in this experiment which play an important role in Serverless function performance.

1. Container Orchestrator
2. Operation Type
3. Programming Language/Runtime.
4. Workload Requests Type
5. Warm/Cold Start Time

4.8.3.2 Dependent Variables

Dependent variable refers the what is being measured and sometimes called output/response. The main goal of the experiment is to measure the performance of the Serverless deployed to different container orchestrators and the following variables represent what we need to study.

1. Response Time
2. Throughout
3. Success Rate

4.8.3.3 Controlled Variables

a variable that is kept constant during the experiment. The following variables represent the settings/variables that should be constant during the whole experiment.

1. **Specification and Number of the Infrastructure Nodes**: Everything related to cloud environment, Operating system, VM node types and the number of nodes should be constant during the experiment.
2. **Function Invocation**: All functions invocation should be triggered from the same network where all Serverless functions deploy to.

4.8.4 Experiment Flow and Scenario

In order to answer the research questions of this thesis, we are going to apply set of scenarios by deploying OpenFaaS Serverless to the selected container orchestrators: Kubernetes, Docker Swarm and Nomad. The base scenarios are divided into four main categories as the following:

1. Computational Scenarios
2. Programming Languages/Runtimes Scenarios
3. Chaining Serverless Functions Scenarios.
4. Warm and Cold Start Scenarios.

There are a set of settings that can be applied and shared between the above scenarios specified as the following:

1. **Sequential Requests**: Requests are generated and sent by *faas-exp* tool sequentially to the deployed functions.
2. **Parallel Requests**: Requests are generated and sent by *faas-exp* tool in parallel using different concurrency levels(5, 10 15, 20, 50) where Serverless function can accept multiple requests at the same time.

3. **Enabled AutoScaling**: AuotScaling can be enabled by OpenFaaS Serverless framework where framework itself can increase the number of replicated function when workloads increases

4. **Disabled AutoScaling**: AuotScaling can be disabled by OpenFaaS Serverless framework where the number of function replicas is set manually (1, 10, 20) to be consistent with number of concurrency levels set before in order to investigate the performance under predefined replicas without enabling auto scaling provided by OpenFaaS.

The following combinations of the above settings can be used together:

1. **Sequential Requests and Disabled AutoScaling**: Autoscaling does not matter here since the number of requests sent is sequential.

2. **Parallel Requests and Enabled AutoScaling**

3. **Parallel Requests and Disabled AutoScaling**

Running the experiment for all scenarios multiple times is very important to make sure that the generated results are valid, eliminate any source of randomness and to improve the accuracy of the collected results. Some previous works did multiple experiment runs varied from 5 - 10 runs like what S. K. Mohanty, G. Premsankar, and M. D. Francesco [105] did on their work for evaluation open source Serverless frameworks. The number of iterations for the experiment study set to 6 runs and it came after conducting some tests and observed that the results were not changing so much after iteration number 6 so that in order to save time and resources, the selected number of iteration was set to 6. Similarly, the total number of submitted request was set to 35000 for each run which is something nearly similar to what others used before.

4.8.4.1 Computational Scenarios

Operation type is considered in this experiment where the following type of operations are going to be implemented using NodeJS programming runtime and deployed to the selected container orchestrators as the following:

1. **I/O Operation**: Function performs write operation to file.

2. **CPU/Memory Operation**: Function performs matrix multiplication

3. **Network Operation**: Function performs file download from the FTP server prepared for this experiment

All of the above scenarios are going to be used with the combinations defined before.

4.8.4.2 Programming Languages/Runtimes Scenarios

Different programming languages runtimes are going to be used using simple function logic where Python, Java, GO and NodeJS are selected. Each language/runtime will be associated with the above combinations settings.

4.8.4.3 Chaining Serverless Functions Scenario

Software applications that use Serverless contain multiple functions that interact with each other and that the reason for adding this scenario where two Serverless functions communicate with each other using RESTful requests. These two functions can be donated as S & D where S represents the source function (caller) and D is the destination function (callee). Both functions written in NodeJS where the destination is a matrix multiplication function and the source function invokes destination. All previous combinations are applied to this scenario.

4.8.4.4 Warm and Cold Start Scenarios

Warm start refers when the function/container is ready to serve requests generated by users as it reuse the available container. However, Cold start refers when no available function/container ready to serve requests which requires an extra container initialization and overhead. The previous combinations are going to be used for Warm and Cold start. One programming language is going to be used which is NodeJS for Warm and Cold start scenarios. Moreover, Its worth to mention that Cold start cases for Nomad are ignored because of technical issues while trying to setup that on Nomad.

4.8.5 Environment Setup

In order to apply the experiment, we need to prepare the required infrastructure and all applications/tools needed to conduct the experiment. The requirements for environment setup specified as the following:

Instance Type	t3a.large
vCPUs	2
RAM	8 GB
Volume (Hard Disk)	15 GB
Operation System	CentoOS Linux 7.6.1810

Table 4.3: Container Orchestrator Nodes Specification

Instance Type	t3a.medium
vCPUs	2
RAM	4 GB
Volume (Hard Disk)	15 GB
Operation System	CentoOS Linux 7.6.1810

Table 4.4: FTP, Invoker and Load Balancer Nodes Specification

1. **Infrastructure Setup**: This includes the cloud infrastructure, virtual machines (nodes) and Operating Systems used for the experiment .

2. **Applications Setup**: This includes the programming languages runtimes/, OpenFaaS Serverless framework, container orchestrators: Kubernetes, Nomad, Docker Swarm, FTP server, *faas-exp* tool, proxy server and OpenFaaS CLI.

4.8.5.1 Infrastructure Setup

In this experiment we are going to use a cloud infrastructure powered by AWS [5]. Morever, the number of nodes used are 6, 3 of them for container orchestrator cluster and the rest for FTP, invoker and proxy nodes. Table 4.3 and Table 4.4 provide details about the required specifications for these nodes.

4.8.5.2 Applications Setup

In order to conduct programming languages runtimes scenarios, we are going to use 4 different programming languages specified in Table 4.5. Moreover, The latest versions of OpenFaaS Serverless framework and CLI are going to be used as specified in Table 4.6 where OpenFaaS will deploy to the selected container orchestrators.

[5]AWS: https://aws.amazon.com/

Programming Languages
Python3
Go1.13
Java8
NodeJS12.13.0

Table 4.5: Programming Languages Runtimes

OpenFaaS Serverless
OpenFaaS Framework 0.18.6
OpenFaaS CLI 0.11.2

Table 4.6: OpenFaaS Serverless

Container orchestrator	**Version**	**Cluster Size**	**Docker version**
Kubernetes	1.16.8	3	19.03.07
Docker Swarm	1.2.6	3	19.03.07
Nomad	0.8.4	3	19.03.07

Table 4.7: Container Orchestrators

Table 4.7 shows the version, size of the cluster for all selected container orchestrators and also the version of docker used. HAProxy(1.5.18) [6] is used as proxy server that balances the requests among the cluster workers. Moreover, Load testing tool is a necessary which helps to generate HTTP requests on behalf of users where Apache Jmeter [7] which is part of *faas-exp*. The number of generated requests will be 35,000 requests. In addition, FTP server is also required for uploading/downloading files where vsftpd [8] is going to be used.

4.8.6 Experiment Tool

As mentioned before on the Data Analysis section, an open source project called *faas-exp* [9] is built for the purpose of automate everything in order to help conducting the experiment, collecting the result and do the comparison for this study. The number of test cases required for this study is quite big where 260 test cases were

[6]HAProxy: `http://www.haproxy.org/`

[7]Apache Jmeter: `https://jmeter.apache.org/`

[8]vsftpd: `https://security.appspot.com/vsftpd.html`

[9]faas-exp: `https://github.com/mabuaisha/faas-exp`

generated for the three container orchestrators as each one is going to have 90 test cases for the 9 deployed functions except Nomad 80. As there are too many test cases to handle, the need for having a tool that can help to automate that instead of doing it manually was needed which played a crucial role to reduce the time and effort for conducting the experiment,collecting results and even analyse them. Moreover, the *faas-exp* is the entry point for this study which makes re-producing the experiment much easier than previous works. Figure 4.3 represents the flow of *faas-exp*.

The functionalities of *faas-exp* can be divided into three main parts:

1. Infrastructure Provisioning & Configuration Management
2. Test Cases Generation.
3. Data Analysis and Visualization.

4.8.6.1 Infrastructure Provisioning & Configuration Management

The Infrastructure is an essential part in this study where virtual machines are required to conduct the experiment. However, managing 18 VMs for the whole experiment is a little bit complex which consumes time and effort where preparing infrastructure environment ready for use requires to setup VMs, operating system, network connectivity, security groups and internet connection to VMs. As experiment will run multiple times for tuning purposes and deleting unused resources, using *faas-exp* reduced the management complexity and helped to automate the provisioning and tearing down resources easily.

On the other hand, infrastructure provisioning is a prerequisite step for another important phase "configuration management" where each container orchestrator cluster has different configuration approach, software packages and dependencies to complete its setup and the remaining nodes FTP, Invoker and Load Balancer also require software packages. The *faas-exp* is using Terraform 0.12.21 [10] for infrastructure provisioning and configuration management where the Terraform code provided by *faas-exp* supports two cloud providers AWS & Openstack.

4.8.6.2 Test Cases Generation

The *faas-exp* provides the ability to conduct all the test cases for each container orchestrator based on the defined performance metrics. Where JMeter load testing

[10]Terraform: https://releases.hashicorp.com/terraform/0.12.21/

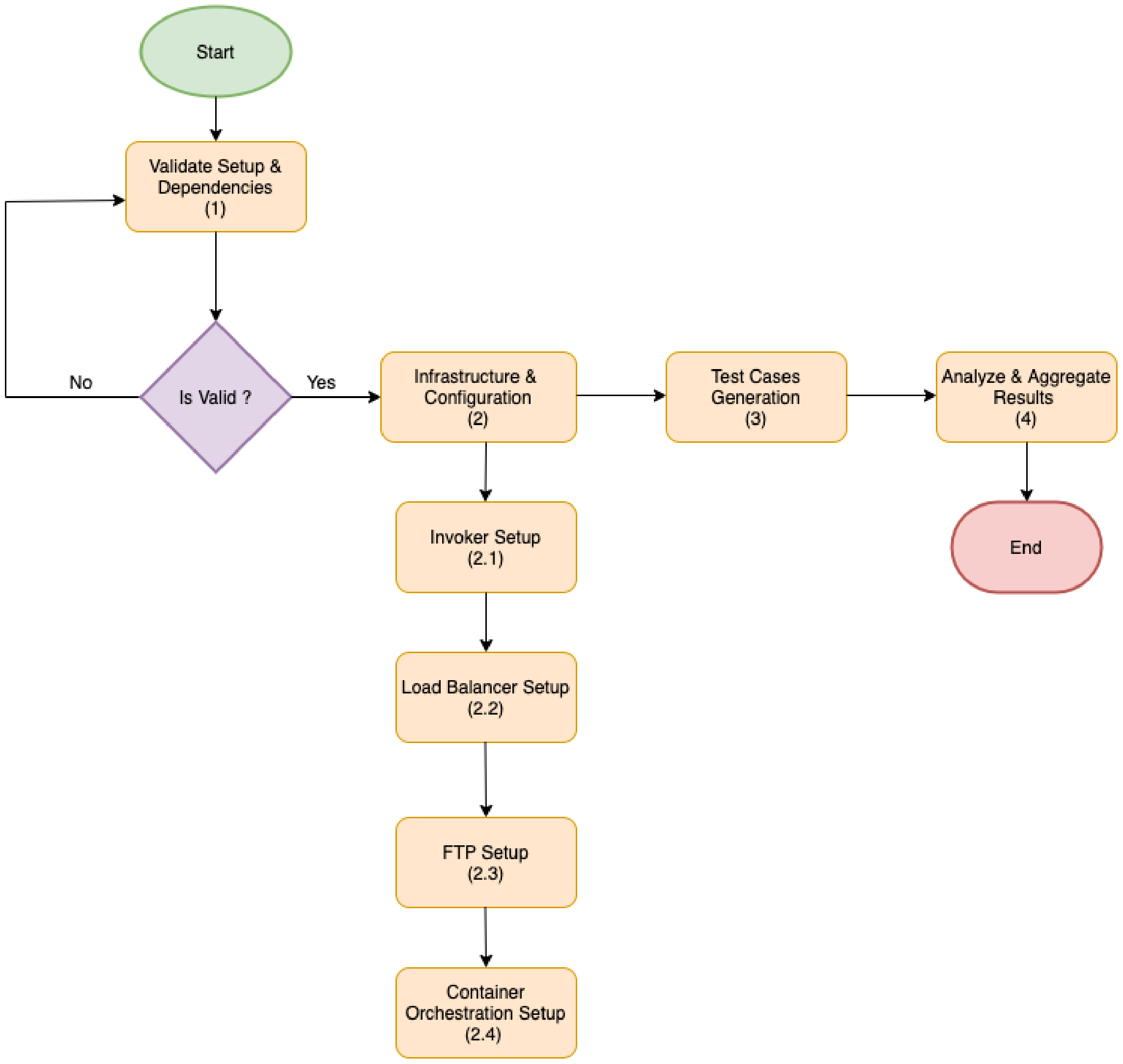

Start
Validate Setup & Dependencies (1)
Is Valid ?
No
Yes
Infrastructure & Configuration (2)
Test Cases Generation (3)
Analyze & Aggregate Results (4)
End
Invoker Setup (2.1)
Load Balancer Setup (2.2)
FTP Setup (2.3)
Container Orchestration Setup (2.4)

Figure 4.3: Faas-Exp Flow

```
experiment:
  server: gateway.openfaas.local
  port: 80
  number_of_runs: 6
  number_of_requests: 35000
  delay_between_runs: 1
  replicas:
    - 1
    - 10
    - 20
  concurrency:
    - 5
    - 10
    - 20
    - 50
```

Listing 1: Snippet 1 Example of faas-exp config.

is integrated with *faas-exp* to help generate the HTTP requests based on the test cases scenarios mentioned before. The snippet yaml file specified on Listing 1 is part of configuration file that represents a common required settings needed to run all tests cases for all Serverless functions. The main common snippet configuration file provides number of total HTTP requests, number of experiment run, delay to add between each runs, number of Serverless function replicas, concurrency levels and finally the information of gateway endpoint. Moreover, the remaining part of configuration file represents required information related to the Serverless function need to be deployed. The yaml snippet file specifed in Listing 2 shows how functions are represented from *faas-exp* perspective where function configuration yaml file, HTTP method and some environment variables are set on the function level. All the configuration files can be found on Appendix **B**

```
functions:
  - name: gofunction
    yaml_path: functions/runtimes-scenarios/go/common/gofunction.yml
    environment:
      read_timeout: 5m5s
      write_timeout: 5m5s
    api:
      uri: function/gofunction
      http_method: POST

  - name: javafunction
    yaml_path: functions/runtimes-scenarios/java/common/javafunction.yml
    environment:
      read_timeout: 5m5s
      write_timeout: 5m5s
    api:
      uri: function/javafunction
      http_method: POST
```

Listing 2: Snippet 2 Example of faas-exp config.

Figure 4.4 shows the flow of single experiment run for all deployed function test cases.

4.8.6.3 Data Analysis and Visualization

The *faas-exp* generates the results for each test case as two type of data: summary & statistics using the JMeter testing tool where summary.jtl & statistics.json files were generated. The summary file contains the raw data related to the information for each request initiated by the JMeter as specified on Table 4.8. On the other hand, the statistics file contains summary of generated test case on single run as the specified on Listing 3.

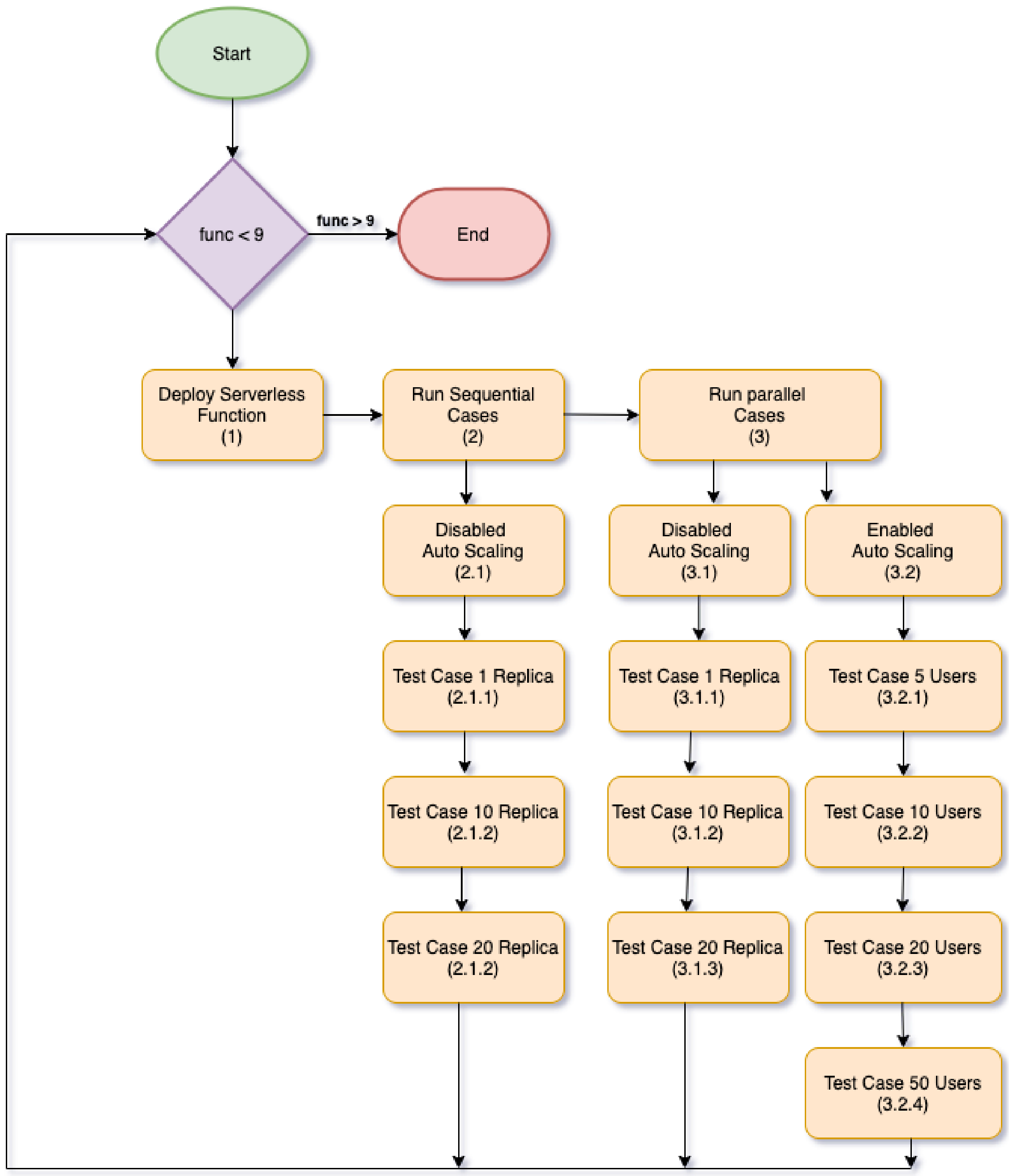

Start
func < 9
func > 9
End
Deploy Serverless Function (1)
Run Sequential Cases (2)
Run parallel Cases (3)
Disabled Auto Scaling (2.1)
Disabled Auto Scaling (3.1)
Enabled Auto Scaling (3.2)
Test Case 1 Replica (2.1.1)
Test Case 1 Replica (3.1.1)
Test Case 5 Users (3.2.1)
Test Case 10 Replica (2.1.2)
Test Case 10 Replica (3.1.2)
Test Case 10 Users (3.2.2)
Test Case 20 Replica (2.1.2)
Test Case 20 Replica (3.1.3)
Test Case 20 Users (3.2.3)
Test Case 50 Users (3.2.4)

Figure 4.4: Faas-Exp Test Cases Flow

```json
"Total" : {
  "transaction" : "Total",
  "sampleCount" : 35000,
  "errorCount" : 0,
  "errorPct" : 0.0,
  "meanResTime" : 77.77742857142798,
  "minResTime" : 9.0,
  "maxResTime" : 1434.0,
  "pct1ResTime" : 125.0,
  "pct2ResTime" : 141.0,
  "pct3ResTime" : 172.0,
  "throughput" : 61.14274034774497,
  "StdDev": 46.06056577053782,
  "receivedKBytesPerSec" : 58.27515606138556,
  "sentKBytesPerSec" : 12.498127162378522
}
```

Listing 3: Statistics JSON Example

timeStamp	elapsed	responseCode	responseMessage	dataType	success	bytes	sentBytes	grpThreads	allThreads	URL	Latency
1589623070767	43	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	37
1589623070769	85	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	85
1589623070770	82	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	75
1589623070775	6	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	6
1589623070780	11	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	11
1589623070781	11	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	11
1589623070782	6	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	6
1589623070787	7	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	7
1589623070792	9	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	9
1589623070793	12	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	12
1589623070797	6	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	6
1589623070799	6	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	6
1589623070801	8	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	8
1589623070805	8	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	7
1589623070806	6	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	6
1589623070807	6	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	6
1589623070812	7	200	OK	text	true	234	194	3	3	http://gateway.openfaas.local/function/gofunction	7
1589623070816	10	200	OK	text	true	234	194	4	4	http://gateway.openfaas.local/function/gofunction	10
1589623070819	12	200	OK	text	true	234	194	4	4	http://gateway.openfaas.local/function/gofunction	12
1589623070822	6	200	OK	text	true	234	194	4	4	http://gateway.openfaas.local/function/gofunction	6

Table 4.8: Sample of Load Testing Response

Additionally, the *faas-exp* has the ability to analyse and aggregate all the summary files in single run iteration for certain Serverless function deployed in all container orchestrators for each test case. The *faas-exp* can generate the results which represents median for all defined metrics and apply the *Wilcoxon* tests for all test cases. Moreover, the *faas-exp* provides the ability to generate figures that compare the results between all container orchestrators of deployed function for certain test case. Figure 4.5 shows the flow of data analysis and visualization.

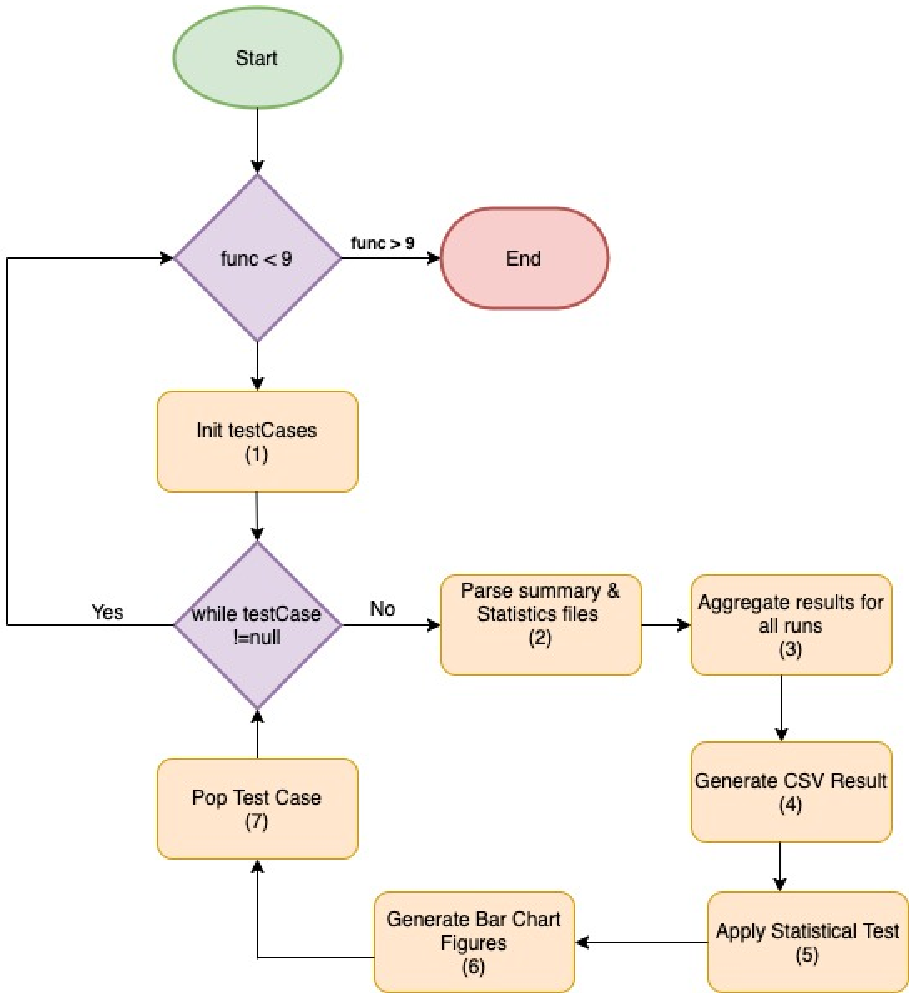

Start
func < 9
func > 9
End
Init testCases (1)
while testCase !=null
Yes
No
Parse summary & Statistics files (2)
Aggregate results for all runs (3)
Generate CSV Result (4)
Apply Statistical Test (5)
Generate Bar Chart Figures (6)
Pop Test Case (7)

Figure 4.5: Faas-Exp Analysis Flow

Chapter 5

Experiment Results

This chapter represents the results of conducting 4 main scenarios as mentioned in Chapter 4 where each scenario is associated with different settings in order to measure the performance of the required metrics on all defined container orchestrators for deploying different Serverless functions using OpenFaas framework. All the results were collected using *faas-exp* where raw data, figures and statistical analysis were generated for all performance metrics response time, throughput and success rate. AWS environment is the infrastructure used for the experiment and in order to avoid and minimize network latency all tests were conducted on the same region *us-east-1*(N. Virginia) and all the requests were invoked within AWS internal infrastructure using the invoker node. All the results of the experiment were gathered within 15 days through 6 runs where the total number of HTTP requests was 35000 for each run. Moreover, for each test case, the *faas-exp* was used in order to aggregate the results of multiple iterations and analyzed them and generate the related figures. The generated results helped to validate the 14 hypotheses set for this study and answered the research questions. Moreover, the results of statistical tests using *Wilcoxon* were collected and aggregated so that we can tell if the results between container orchestrators are statistically significance and did not occur randomly or by chance. We used two symbols ▲ ▽ to indicate that results are statistically significance *p-value* $< 5\%$ between each container orchestrator for a given test case. The black triangle means that if we have a relationship between X & Y then X performance is better than Y. However, down triangle means that Y performance is better than X. On the other hand, we used dash symbol (–) to indicate that results for a given test case are not statistically significance and there is no difference between using$X$ & Y.

5.1 Computation Scenarios

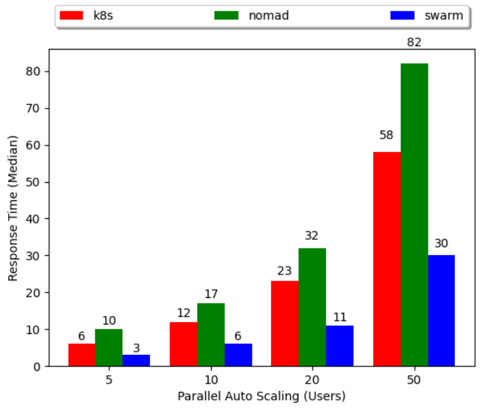
In this section the results of three main computation scenarios for I/O, CPU/Memory and Network operations were collected and all functions were written in NodeJS. The results of the computation test cases addresses the H_7- H_9 hypotheses about the relationship between the operation type of the deployed Serverless function for each performance metric and container orchestrators.

5.1.1 I/O Operation

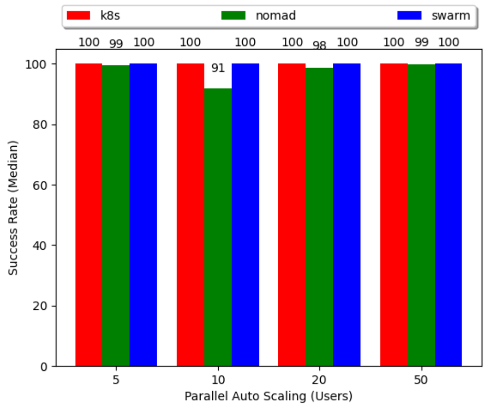
This refers to the results of the Serverless function that performs I/O operation by accepting a large string and dump it to the file system of the deployed function. The results of the I/O operation for parallel and sequential test cases are shown in Figures 5.1, 5.2 and 5.3.

Based on Figure 5.1, the parallel test cases with auto scaling enabled were generated and its clear that Docker Swarm performed better than others. Response time was increasing as the number of users increased between all container orchestrators which was something expected as the load increased. Docker Swarm had the smallest response time among others. Success rate for both K8S and Docker Swarm were the same with 100% success rate for all cases. However, Nomad had small percentage of errors varied from 1% - 9%. Moreover, Docker Swarm achieved the best throughput with 2x than K8S and 3x than Nomad as the highest value was observed when concurrency level was 20 (1592 requests/second).

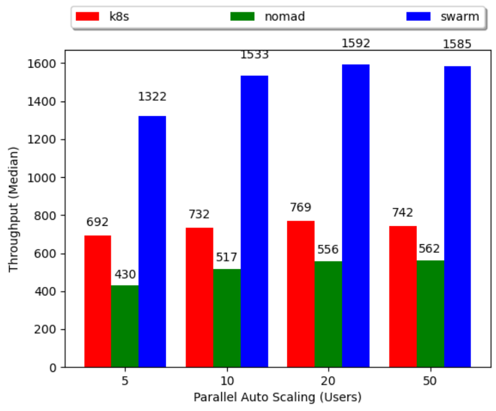
Figure 5.2 represents the cases where parallel workloads were generated using fixed number of concurrency level 15 as the auto scaling was disabled for the I/O Serverless function and the number of replicas set manually (1, 10, 20). Generally, Docker Swarm performed better than other orchestrators. Response time started with higher values for all orchestrators as there was only one replica serving all the requests. Its noticeable that using 10 replicas had impact on response time for all orchestrators. However, using 20 replicas had slightly increase in response time for all orchestrators. The success rate was 100% for the initiated requests among all orchestrators. Moreover, throughput behaved the same way as response time where on replica 20 there was a slightly decrease in throughput where theoretically it should be increased as the number of replicas were increased. Docker Swarm performed 1.5x faster than K8S and 3x than Nomad in terms of response time and throughput.



(a) Response Time

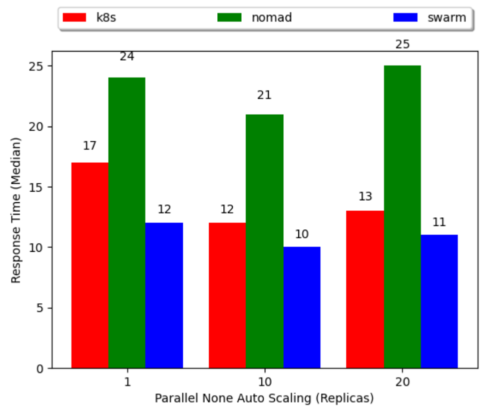


(b) Success Rate

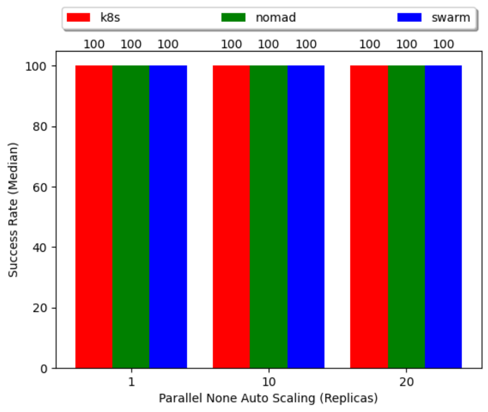


(c) Throughput

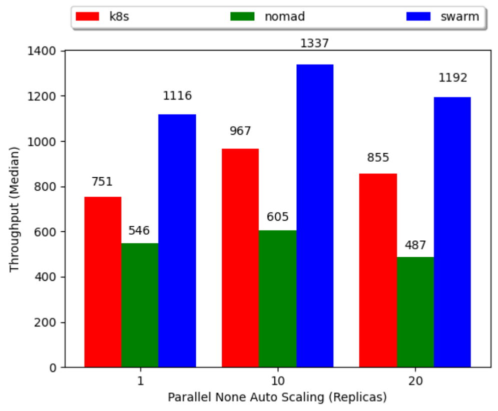
Figure 5.1: I/O Parallel Auto Scaling Enabled



(a) Response Time

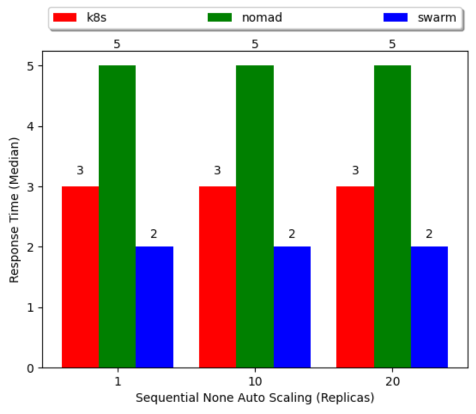


(b) Success Rate

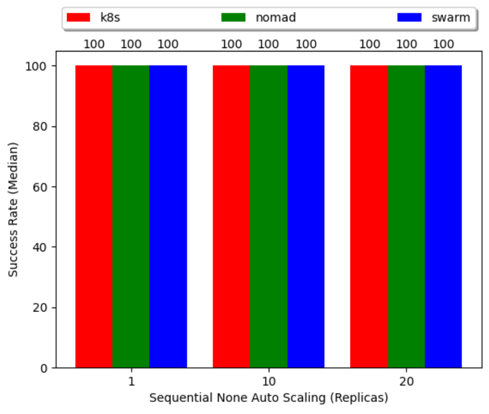


(c) Throughput

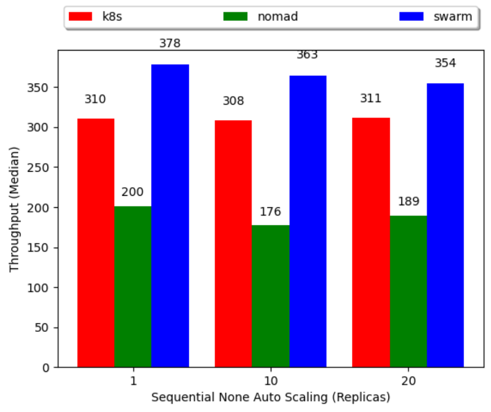
Figure 5.2: I/O Parallel Auto Scaling Disabled



(a) Response Time



(b) Success Rate



(c) Throughput

Figure 5.3: I/O Sequential Auto Scaling Disabled

Figure 5.3 represents test cases where sequential workloads with auto scaling disabled were generated using one user at a time where different number of replicas was set. Nearly the response time for both Docker Swarm and K8S were the same and Nomad had the worst result. On the other hand, the success rate for all orchestrators were identical. However, in general Docker Swarm performed better than others in terms of throughput. Its clear from the results that increasing the number of replicas for sequential workloads did not have noticeable impacts on the metrics as the replicas may not fully utilized because of the sequential workload requests.

Tables 5.1, 5.2 and 5.3 specify the results of applying *Wilcoxon* tests for all i/o test cases between each container orchestrator.

	nomad	**swarm**
k8s	▲ ▲ ▲ ▲	▽ ▽ ▽ ▽
nomad		▽ ▽ ▽ ▽

(a) Response Time

	nomad	**swarm**
k8s	▲ ▲ ▲ ▲	▽ ▽ ▽ ▽
nomad		▽ ▽ ▽ ▽

(b) Throughput

	nomad	**swarm**
k8s	▲ ▲ ▲ –	– – – –
nomad		▽ ▽ ▽ –

(c) Success Rate

Table 5.1: Wilcoxon I/O Parallel Auto Scaling Enabled

Table 5.1 represents the *Wilcoxon* results for parallel workloads with auto scaling enabled that covers four the concurrency levels (5, 10, 20, 50). Response time results were statistically significant between all container orchestrators where K8S had better results than Nomad for all concurrency levels based on the black triangles. Moreover, Docker Swarm performed better than both K8S & Nomad which aligned with the results we obtained before. Moreover, throughput results were statistically significant where K8S performed better than Nomad and Docker Swarm had better result than all of them. Finally Success rate, also showed the same behavior and confirmed the results we obtained are aligned with our statistical tests where K8S and Docker Swarm had better results than Nomad and the results for Docker Swarm and K8S were them same which explains the results we got from applying Wilcoxon tests as there was no statistical significant result for success rate for them.

	nomad	**swarm**
k8s	▲ ▲ ▲	▽ ▽ ▽
nomad		▽ ▽ ▽

(a) Response Time

	nomad	**swarm**
k8s	▲ ▲ ▲	▽ ▽ ▽
nomad		▽ ▽ ▽

(b) Throughput

	nomad	**swarm**
k8s	– – –	– – –
nomad		– – –

(c) Success Rate

Table 5.2: Wilcoxon I/O Parallel Auto Scaling Disabled

Table 5.2 represents the *Wilcoxon* results for parallel workloads with auto scaling disabled using three different replica settings (1, 10, 20). The statistical tests for both response time and throughput aligned with our results regarding the preferblility of Docker Swarm on both K8S and Nomad. The comparison between K8S & Nomad showed that results were statistically significant and K8S performed better than Nomad based on the direction of the black triangle for both response time and throughput. Moreover, the comparison between K8S and Docker Swarm showed that the results were statistically significant but Docker Swarm had better results than K8S and the same thing applied for the relationship between Docker Swarm and Nomad. Finally, the results for success rate showed that the results were not statistically significant between all container orchestrators as they have the same success rate.

	nomad	**swarm**
k8s	▲ ▲ ▲	▽ – –
nomad		▽ ▽ ▽

(a) Response Time

	nomad	**swarm**
k8s	▲ ▲ ▲	▽ ▽ ▽
nomad		▽ ▽ ▽

(b) Throughput

	nomad	**swarm**
k8s	– – –	– – –
nomad		– – –

(c) Success Rate

Table 5.3: Wilcoxon I/O Sequential Auto Scaling Disabled

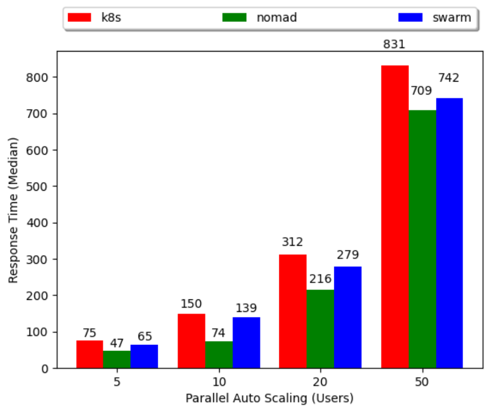
Table 5.3 represents the *Wilcoxon* results for sequential workloads with auto scaling disabled where both response time and throughput results for all three replica settings had p-values < 5% based on the triangle symbols where Docker Swarm had best results in all replica settings. However, its noticeable that for response time the replicas (10, 20) had - values which means that results were not statistically significant. Finally, Success rate results of all container orchestrators pairs for all replica settings were identical which explains the results why the success rate were not statistically significant.

5.1.2 CPU/Memory Operation

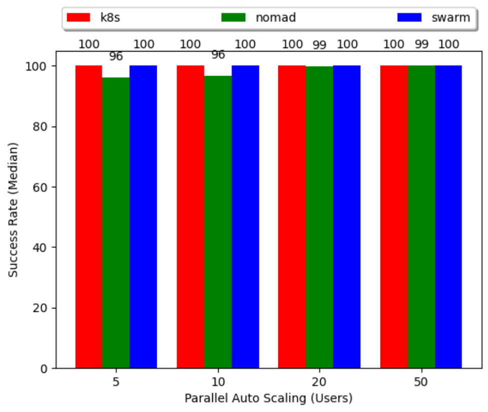
The CPU/Memory operation refers to the result of Serverless function that performs multiplication of two dimensional matrices with different sizes varies from 10 X 10 to 200 X 200 which was the same function used by Back and V. Andrikopoulos [89]. The Matrix function was written in NodeJS that covered both parallel and sequential test cases. Figures 5.4, 5.5 and 5.6 represent the results for both parallel and sequential test cases.

Figure 5.4 represents the generated results of parallel workload with auto scaling enabled. Nomad preformed better than others in terms of response time and throughput in all cases. However, K8S & Docker Swarm had 100% success rate whereas Nomad had small percentage of errors 1%- 4%. There was an increasing spike for response time with 50 concurrency level and its acceptable because the number of concurrency levels increased and the complexity of the function was relatively high and that explains also how the result of this function is compared to previous function(I/O) as range of values were higher for response time and lower in throughput.

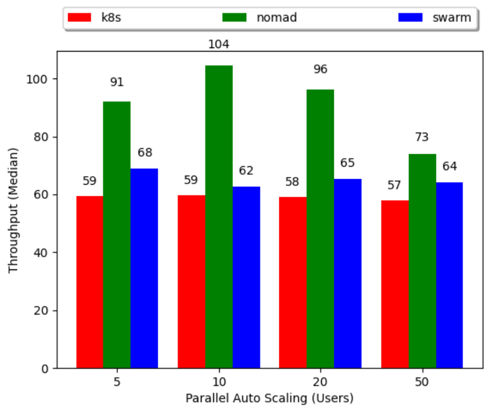
Figure 5.5 shows how the results are presented using parallel workloads with auto scaling disabled. The concurrency level set to 15 for all test cases where three different values of replicas were used (1, 10, 20). Generally, the Docker Swarm performed better than others in term of response time and throughput. Moreover, the success rate was identical for all container orchestrators. Using only one replica had clear impacts on both response time and throughput for all container orchestrators as one replica will not be able to serve all the concurrent requests as it should be. On the other hand, increasing the number of replicas showed an enchantment for both response time and throughput where Docker Swarm achieved the highest throughput (223 requests/second) at replica 10 and the lowest response time (37ms) at replica



(a) Response Time

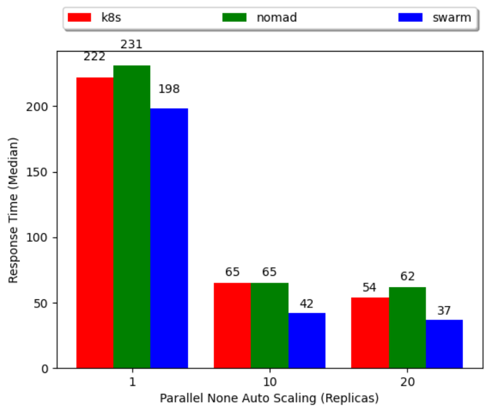


(b) Success Rate

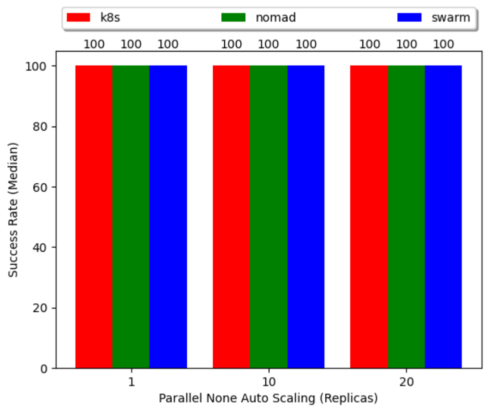


(c) Throughput

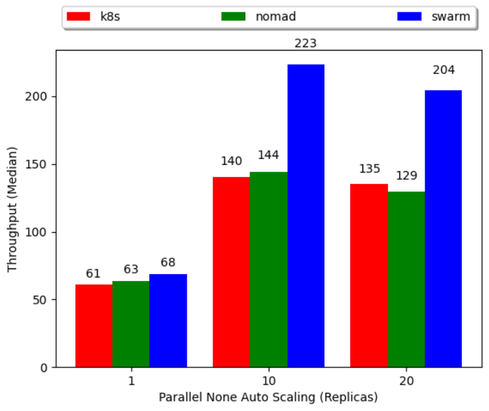
Figure 5.4: Matrix Multiplication Parallel Auto Scaling Enabled



(a) Response Time



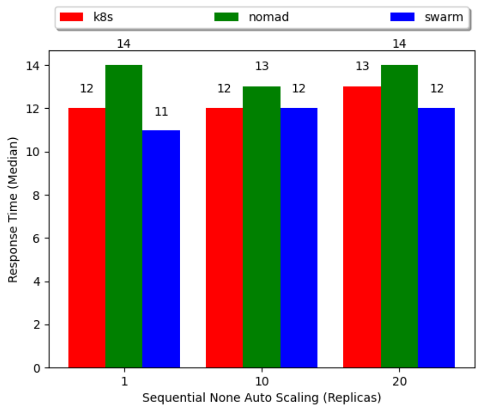
(b) Success Rate



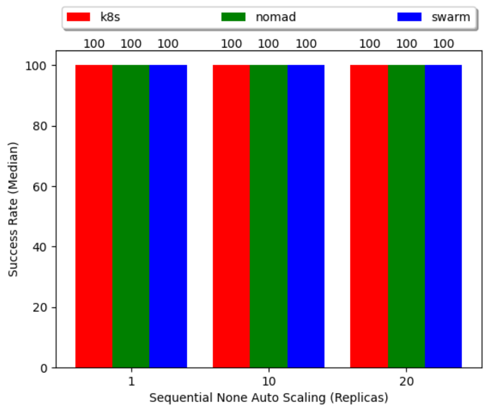
(c) Throughput

Figure 5.5: Matrix Multiplication Parallel Auto Scaling Disabled

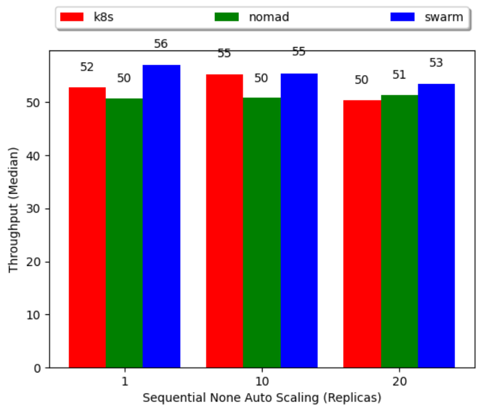
20.



(a) Response Time



(b) Success Rate



(c) Throughput

Figure 5.6: Matrix Multiplication Sequential Auto Scaling Disabled

Figure 5.6 shows the results of sequential workloads with auto scaling disabled where the Docker Swarm again performed better than others in term of response time and throughput. However, the success rate was identical for all container orchestrators. There was no impact for the number of increasing replicas when handling the sequential workloads as there was only one request at a time which made other replicas un-utilized. Moreover, result values of response time and throughput were less than parallel workloads results.

	nomad	**swarm**
k8s	▽ ▽ ▽ ▽	▽ ▽ ▽ ▽
nomad		– ▲ ▲ –

(a) Response Time

	nomad	**swarm**
k8s	▽ ▽ ▽ ▽	▽ ▽ ▽ ▽
nomad		– ▲ – –

(b) Throughput

	nomad	**swarm**
k8s	▲ ▲ ▲ ▲	– – – –
nomad		▽ ▽ ▽ ▽

(c) Success Rate

Table 5.4: Wilcoxon Matrix Multiplication Parallel Auto Scaling Enabled

Table 5.4 represents the *Wilcoxon* results for parallel workloads with auto scaling enabled using four concurrency levels (5, 10, 20, 50). The results for response time showed that both Docker Swarm and Nomad had better results than K8S where the results for K8S with Docker Swarm and Nomad were statistically significant as p-values $< 5\%$. However, the results of (Nomad, Docker Swarm) pair were statistically significant on concurrency levels (10, 20). On the other hand, the results for throughput showed that both Docker Swarm and Nomad had better results than K8S where the results for (K8S, Nomad) & (K8S, Docker Swarm) were statistically significant with p-values $< 5\%$. The results for (Nomad, Docker Swarm) pair were statistically significant on concurrency level (10) which was close to the results we obtained before. Finally, the results for success rate were statistically significant for (K8S, Nomad) & (Nomad, Docker Swarm) since Nomad had some error rate whereas K8S and Docker Swarm had 100% success rate which explains the *Wilcoxon* results.

	nomad	**swarm**
k8s	– – ▲	▽ ▽ ▽
nomad		▽ ▽ ▽

(a) Response Time

	nomad	**swarm**
k8s	▽ – ▲	▽ ▽ ▽
nomad		▽ ▽ ▽

(b) Throughput

	nomad	**swarm**
k8s	– – –	– – –
nomad		– – –

(c) Success Rate

Table 5.5: Wilcoxon Matrix Multiplication Parallel Auto Scaling Disabled

Table 5.5 represents the *Wilcoxon* results for parallel workloads with auto scaling disabled using three different replica settings (1, 10, 20). The results of response time were statistically significant for all pairs across all replica settings except for (K8S, Nomad) pair on replicas (1, 10) which means the performance for Serverless function using these replicas under Nomad or K8S will have no impact on response time. Moreover, the results of (K8S, Docker Swarm) & (Nomad, Docker Swarm) pairs were statistically significant as Docker Swarm had best results than others. The throughput results showed that all container orchestrators pairs using all replica settings were statistically significant except of (K8S, Nomad) pair on replica 10. The results of (K8S, Nomad) pair were statistically significant using replica settings (1, 20) where Nomad had better results on replica 1 and K8S performed better on replica 20. On the other hand, the results of (K8S, Docker Swarm) & (Nomad, Docker Swarm) pairs were statistically significant with p-values $< 5\%$. Finally, the results of success rate on all pairs were not statistically significant since the success rate were identical in all replicas for all container orchestrators.

	nomad	**swarm**
k8s	– – ▲	▽ ▽ ▽
nomad		▽ ▽ ▽

(a) Response Time

	nomad	**swarm**
k8s	▽ – ▲	▽ ▽ ▽
nomad		▽ ▽ ▽

(b) Throughput

	nomad	**swarm**
k8s	– – –	– – –
nomad		– – –

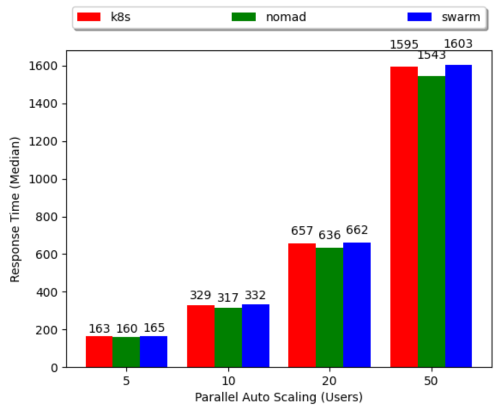
(c) Success Rate

Table 5.6: Wilcoxon Matrix Multiplication Sequential Auto Scaling Disabled

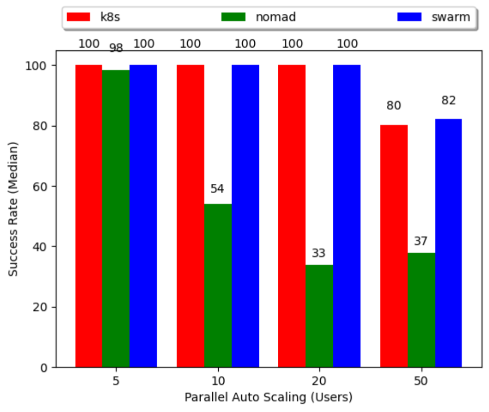
Table 5.6 represents the *Wilcoxon* results for sequential workloads with auto scaling disabled using the same replica settings used before. The results of response time, throughput were statistically significant with p-values $< 5\%$ and similar to the results we obtained on previous parallel test cases with disabled auto scaling. Finally, the results for success rate in all test case across all pairs are not statistically significant.

5.1.3 Network Operation

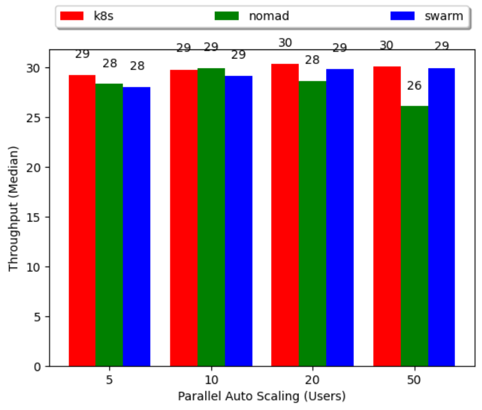
The network operation refers to the result of Serverless function that preforms downloading file from FTP server that reside on the same network of container orchestrators to reduce any overhead and network latency. The function was written in NodeJS which covered both parallel and sequential test cases. Figures 5.7, 5.8 and 5.9 specify the results collected.



(a) Response Time



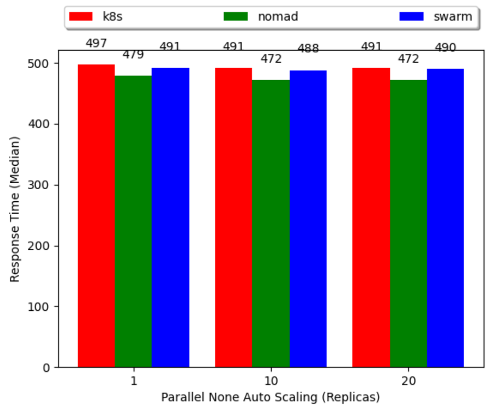
(b) Success Rate



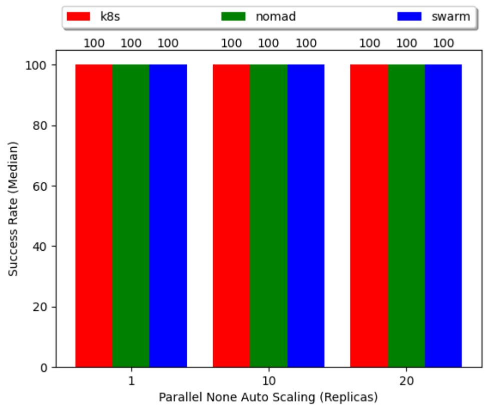
(c) Throughput

Figure 5.7: Network Parallel Auto Scaling Enabled

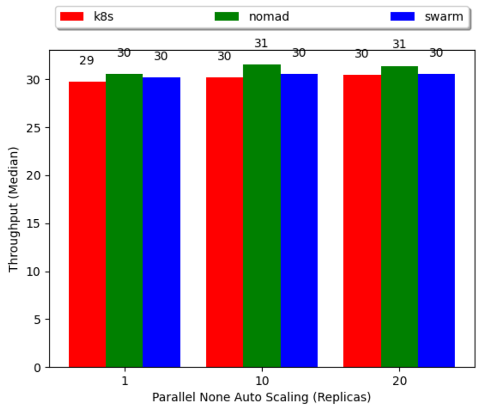
Figure 5.7 represents the collected results of parallel workloads with auto scaling enabled. Nomad performed better than others in term of response time. However, Nomad had the worst value for success rate where error percentage was the highest and varied from 46% - 67% specially with concurrence levels (10, 20, 50) and both K8S and Docker Swarm had small error percentage with concurrency level 50. On the other hand, the throughput values in all concurrency levels for all container orchestrators were closed to each other with preferability to K8S. The parallel workloads with increasing concurrency levels had noticeable impacts on the success rate especially for Nomad and this could be because of the Nomad provider was not able to initiated enough number of replicas to serve all the requests which explains the low success rate.



(a) Response Time



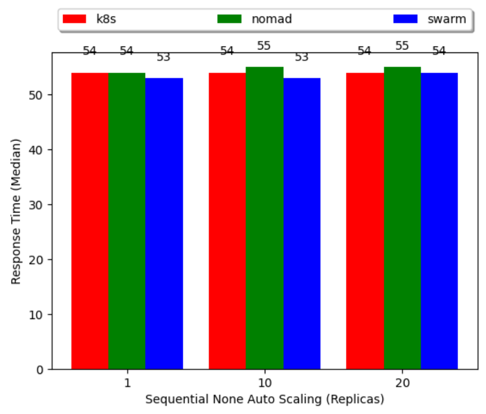
(b) Success Rate



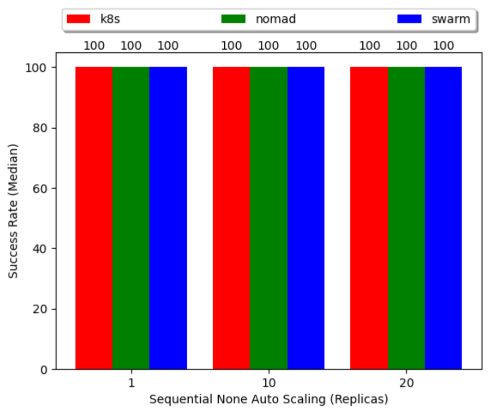
(c) Throughput

Figure 5.8: Network Parallel Auto Scaling Disabled

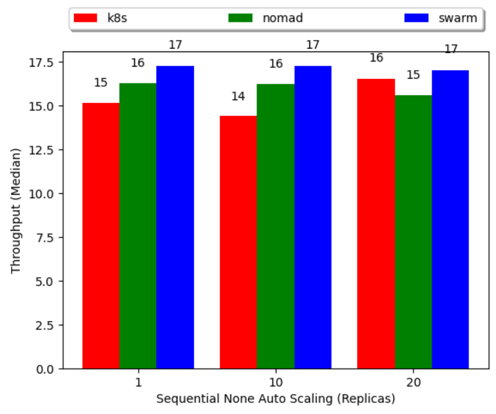
Figure 5.8 shows the results of parallel workloads with auto scaling disabling where the replicas were set manually. The same as other functions, the concurrency level was fixed and set to 15 for all test cases. Nomad performed slightly better than others in term of response time and throughput where they almost close to each other. The success rate was identical for all container orchestrators with 100% success rate as the number of replicas were able to serve all the required requests which connected to the FTP server and download a file.



(a) Response Time



(b) Success Rate



(c) Throughput

Figure 5.9: Network Sequential Auto Scaling Disabled

Figure 5.9 shows the results of sequential workloads with auto scaling disabled. The concurrency level for this case was set to one request at a time and its obvious from the result above that increasing the number of replicas had almost zero impact on the response time and success rate as the number of replicas were not utilized. Generally, Docker Swarm performed better than others in terms of throughput. However, the success rate of all container orchestrators are the same and the response time values nearly the same and close to each other.

	nomad	**swarm**
k8s	▽ ▽ ▽ ▽	▲ ▲ ▲ ▲
nomad		▲ ▲ ▲ ▲

(a) Response Time

	nomad	**swarm**
k8s	▲ – – ▲	▲ ▲ ▲ ▲
nomad		– ▲ – ▽

(b) Throughput

	nomad	**swarm**
k8s	▲ ▲ ▲ ▲	– – – ▽
nomad		– ▽ ▽ ▽

(c) Success Rate

Table 5.7: Wilcoxon Network Parallel Auto Scaling Enabled

Table 5.7 represents the *Wilcoxon* results for parallel workloads with auto scaling enabled using four concurrency levels (5, 10, 20, 50) where response time results were statistically significant between all container orchestrators pairs where p-values for all concurrency levels $< 5\%$ which aligned with our previous results. It's clear from the triangles direction that both K8S & Nomad performed better than Docker Swarm. On the other hand, Nomad performed better than K8S. The throughput results were statistically significant for all concurrency levels of (K8S, Docker Swarm) pair where K8S performed better than Docker Swarm. However, the results of other pairs (K8S, Nomad), (Nomad, Docker Swarm) were statistically significant on (5, 50) & (10, 50) replica settings respectively. Moreover, the success rate results were statistically significant for all container orchestrators pairs. The result of (K8S, Nomad) pair were statistically significant for all concurrency levels because of the high differences between their success rates. However results of (Nomad, Docker Swarm) were statistically significant in all concurrency levels except on level 5. Finally, the results for pair (K8S, Docker Swarm) were statistically significant on concurrency level 50 where Docker Swarm performed better thank K8S.

Table 5.8 represents the *Wilcoxon* results for parallel workloads with auto scaling disabled using three different replica settings (1,10,20). The response time results were statistically significant between all container orchestrators pairs for all replica settings with p-values $< 5\%$ except on (K8S, Docker Swarm) pair using replica 20.

	nomad	**swarm**
k8s	▽ ▽ ▽	▽ ▽ –
nomad		▲ ▲ ▲

(a) Response Time

	nomad	**swarm**
k8s	▽ ▽ ▽	▽ ▽ –
nomad		– ▲ ▲

(b) Throughput

	nomad	**swarm**
k8s	– – –	– – –
nomad		– – –

(c) Success Rate

Table 5.8: Wilcoxon Network Parallel Auto Scaling Disabled

The response time results showed that Nomad performed better than K8S & Docker Swarm on all replica settings. The same thing applied for throughput where the results of all pairs tests were statistically significant except on replica 20 for pair (K8S, Docker Swarm) and on replica 1 for pair (Nomad, Docker Swarm). Finally, the results of success rate for all pairs were not statistically significant where they had the same success rate across all replica settings.

	nomad	**swarm**
k8s	– ▲ ▲	▽ – –
nomad		▽ ▽ ▽

(a) Response Time

	nomad	**swarm**
k8s	▽ ▽ ▲	▽ ▽ ▽
nomad		▽ ▽ ▽

(b) Throughput

	nomad	**swarm**
k8s	– – –	– – –
nomad		– – –

(c) Success Rate

Table 5.9: Wilcoxon Network Sequential Auto Scaling Disabled

Table 5.9 represents the *Wilcoxon* results for sequential workloads using the same replica settings used before. The results were statistically significant with p-values $<$ 5% for response time and throughput. The results of response time were statistically significant on all container orchestrators pairs except on replica 1 for (K8S, Nomad) pair and on replicas (10, 20) for (K8S, Docker Swarm) which aligned with the results we obtained before. On the other hand, the results for throughput were statistically significant for all replicas across all pairs where Docker Swarm performed better than others and K8S performed better than Nomad on replicas (1,10). Finally, the results for success rate of all container orchestrators pairs were not statistically significant as they all have 100% success rate across all replica settings.

5.2 Programming Languages/Runtimes Scenarios

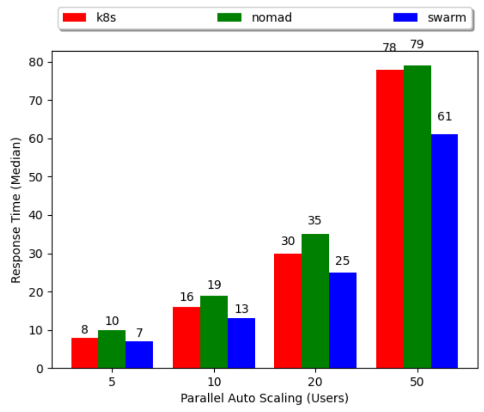
This section represents the results of four Serverless functions which were written using three programming languages and JavaScript runtime environment (NodeJS) to compare the defined metrics for each programming language. The results of the programming languages test cases addresses the H_{11}-H_{12} hypotheses about the relationship between the programming language of the deployed Serverless function and container orchestrator.

5.2.1 Python Function

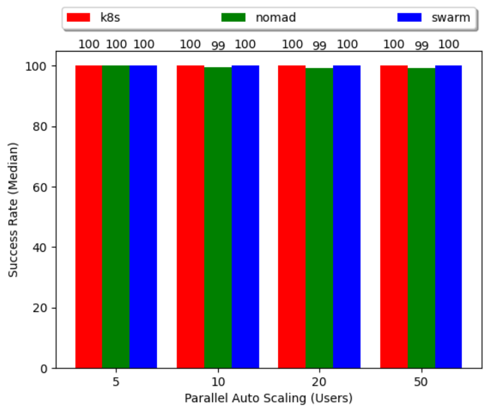
This scenario represents a Serverless function that was written in Python 3 which returns a simple message on call invocation. This scenario covered both parallel and sequential test cases as specified in Figures 5.10, 5.11 and 5.12.

Figure 5.10 shows the results of the parallel workloads requests with auto scaling enabled. Docker Swarm relatively performed better than others in term of response time and throughput. Success rate for both Docker Swarm and K8S were identical whereas Nomad had a small error percentage 1%. Response time was increased as the number of concurrency levels increased which was expected because of the high loads where Docker Swarm had the lowest response time (7ms). Moreover, the throughput was increased with increasing the number of concurrency levels as the number of replicas will be increased based on the requests load where Docker Swarm achieved the highest throughput (791 requests/second) on concurrency level 50.

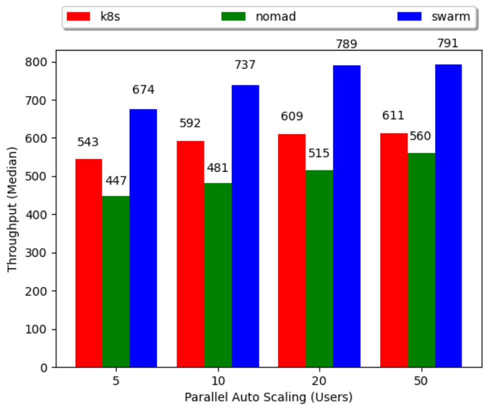
Figure 5.11 represents the results of parallel workloads requests with auto scaling disabled where number of concurrency level set to 15 and used different number of replicas (1, 10, 20). The Docker Swarm performed better than others in term of response time and throughput. The success rate was identical for all container orchestrators. On the other hand, the highest throughput achieved when the number of replicas was set to 10 where the Docker Swarm had the best result (1243 requests/second). Comparing the results with enabled auto scaling its obvious that response time and throughput with 10 replicas had better performance.



(a) Response Time

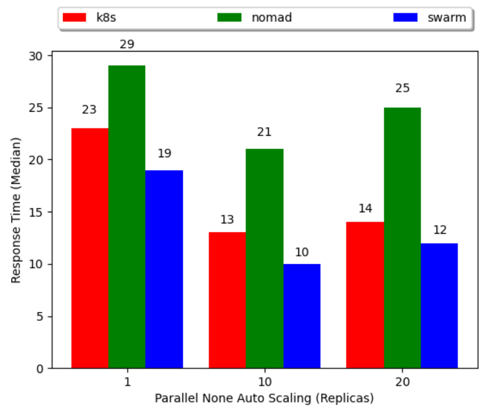


(b) Success Rate

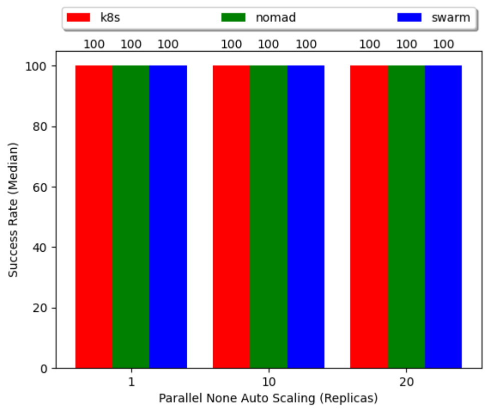


(c) Throughput

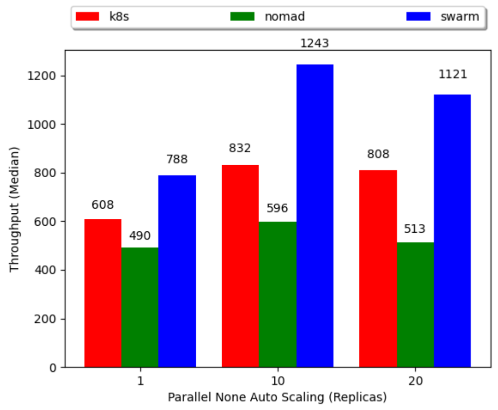
Figure 5.10: Python Parallel Auto Scaling Enabled



(a) Response Time

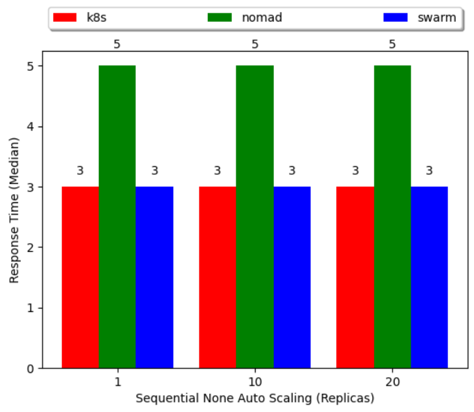


(b) Success Rate

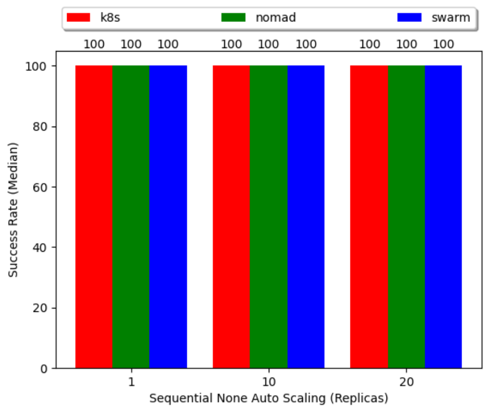


(c) Throughput

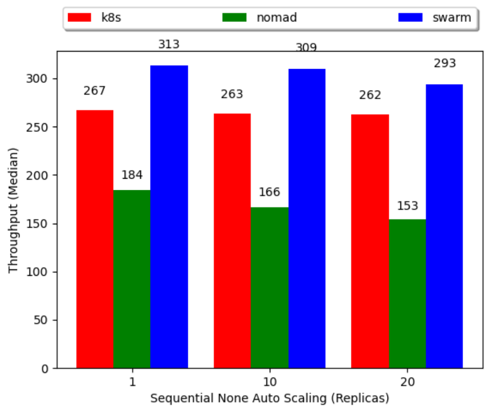
Figure 5.11: Python Parallel Auto Scaling Disabled



(a) Response Time



(b) Success Rate



(c) Throughput

Figure 5.12: Python Sequential Auto Scaling Disabled

Figure 5.12 shows the results of sequential workloads with auto scaling disabled. 3 different replicas number used (1, 10, 20) by applying a sequential workloads where one request a time generated by the *faas-exp*. Based on these results, increasing the number of replicas had nearly no impact on response time, throughput and success rate as it seems only one replica was enough to serve all requests. Response time was identical for both Docker Swarm and K8S and had a higher values on Nomad. Moreover, success rate was identical for all container orchestrators. However, Docker Swarm performed better than others in term of throughput in all replicas settings.

	nomad	**swarm**
k8s	▲ ▲ ▲ –	▽ ▽ ▽ ▽
nomad		▽ ▽ ▽ –

(a) Response Time

	nomad	**swarm**
k8s	▲ ▲ ▲ ▲	▽ ▽ ▽ ▽
nomad		▽ ▽ ▽ ▽

(b) Throughput

	nomad	**swarm**
k8s	– ▲ – ▲	– – – –
nomad		– ▽ ▽ ▽

(c) Success Rate

Table 5.10: Wilcoxon Python Parallel Auto Scaling Enabled

Table 5.10 represents the *Wilcoxon* results for parallel workloads using four concurrency levels (5, 10, 20, 50). The results for response time, throughput and success rate were statistically significant with p-values $< 5\%$. The response time results were statistically significant for all container orchestrators pairs using all concurrency levels except on level 50 for (K8S, Nomad) & (Nomad, Docker Swarm). The response time for Docker Swarm performed better than others which aligned with the results we obtained before. On the other hand, the throughput results of all container orchestrators pairs were statistically significant for all concurrency levels where Docker Swarm and K8S performed better than Nomad. Finally, the results for success rate were statistically significant for (K8S, Nomad) & (Nomad, Docker Swarm) only since Nomad had some error rate which explain the result we obtained before.

Table 5.11 represents the *Wilcoxon* results for parallel workloads with auto scaling disabled using three replica settings (1, 10, 20) where both response time and throughput results were statistically significant for all container orchestrators pairs under all replica settings for p-values $< 5\%$. the results of K8S & Docker Swarm performed better than Nomad which aligned with the results we obtained before. However, the results of success rare for all container orchestrators pairs using all

	nomad	**swarm**
k8s	▲ ▲ ▲	▽ ▽ ▽
nomad		▽ ▽ ▽

(a) Response Time

	nomad	**swarm**
k8s	▲ ▲ ▲	▽ ▽ ▽
nomad		▽ ▽ ▽

(b) Throughput

	nomad	**swarm**
k8s	– – –	– – –
nomad		– – –

(c) Success Rate

Table 5.11: Wilcoxon Python Parallel Auto Scaling Disabled

replica settings were not statistically significant as the success rate were 100% for all container orchestrators.

	nomad	**swarm**
k8s	▲ ▲ ▲	▽ ▽ –
nomad		▽ ▽ ▽

(a) Response Time

	nomad	**swarm**
k8s	▲ ▲ ▲	▽ ▽ ▽
nomad		▽ ▽ ▽

(b) Throughput

	nomad	**swarm**
k8s	– – –	– – –
nomad		– – –

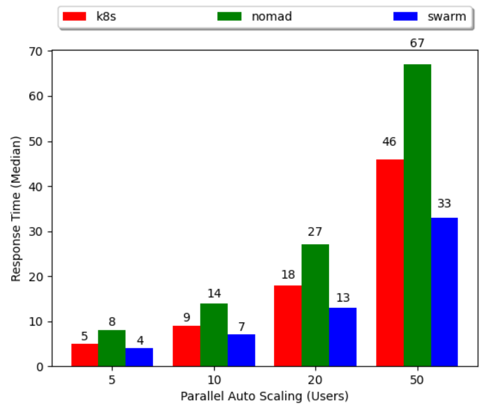
(c) Success Rate

Table 5.12: Wilcoxon Python Sequential Auto Scaling Disabled

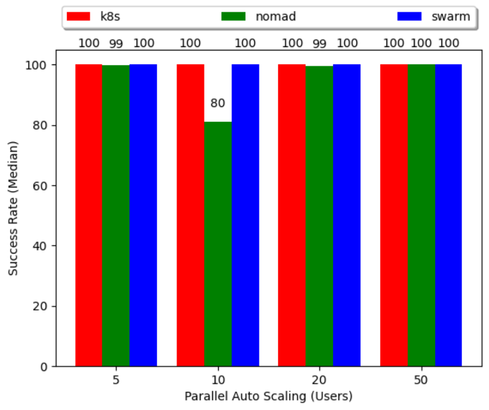
Table 5.12 represents the *Wilcoxon* results for sequential workloads using the same replica settings used before where response time and throughput were statistically significant with p-values $< 5\%$ for all container orchestrators pairs in all replicas except on (K8S, Docker Swarm) on replica 20 for response time results. Finally, the results for success rate were not statistically significant for all container orchestrators pairs using all replica settings as they had the same success rate results.

5.2.2 NodeJS Function

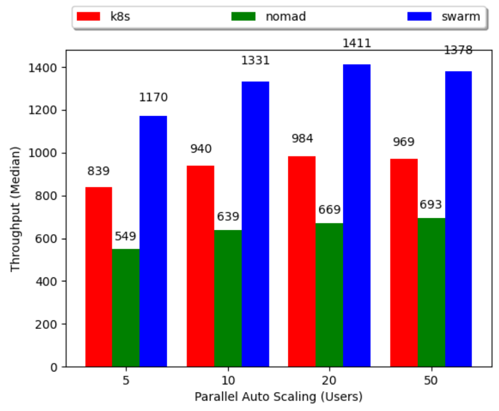
This scenario represents a Serverless function that was written in NodeJS 12 that returns a simple message on call invocation. It covered both parallel and sequential test cases as specified on Figures 5.13, 5.14 and 5.15.



(a) Response Time



(b) Success Rate

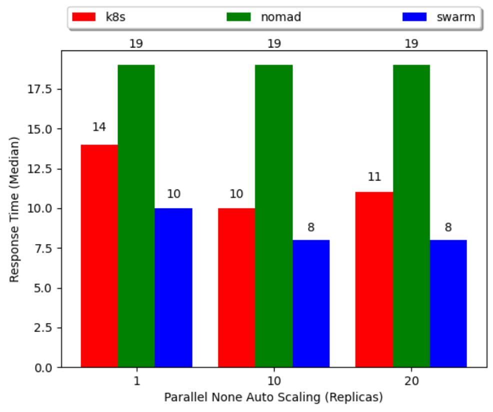


(c) Throughput

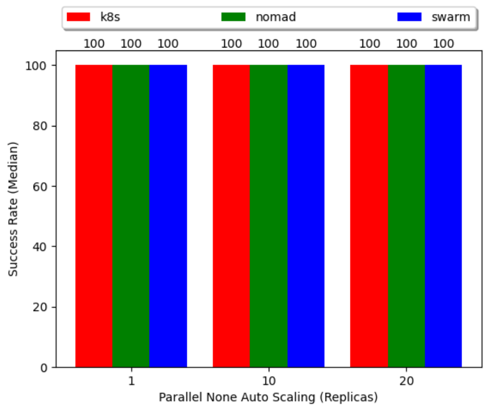
Figure 5.13: NodeJS Parallel Auto Scaling Enabled

Figure 5.13 shows the generated results of parallel workloads with auto scaling enabled. The Docker Swarm performed better in terms of response time and throughput than others. Success rate was identical for both Docker Swarm and K8S whereas Nomad had an noticeable error rate 20% on concurrency level 10 and small 1% on (5, 20) levels. Docker Swarm had 2X throughput than Nomad and nearly

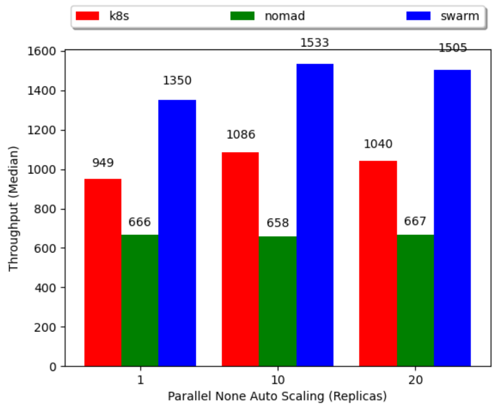
1.5X better than K8S where the highest value was achieved on concurrency level 20 (1411 requests/second). Moreover, Docker Swarm achieved the lowest response time (4ms) at concurrency level 5. Comparing the results of Python function with NodeJS, NodeJS performed better in terms of all metrics.



(a) Response Time



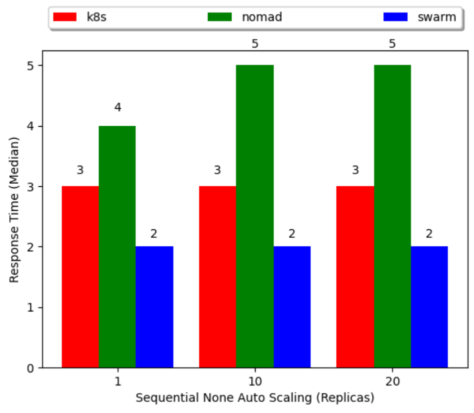
(b) Success Rate



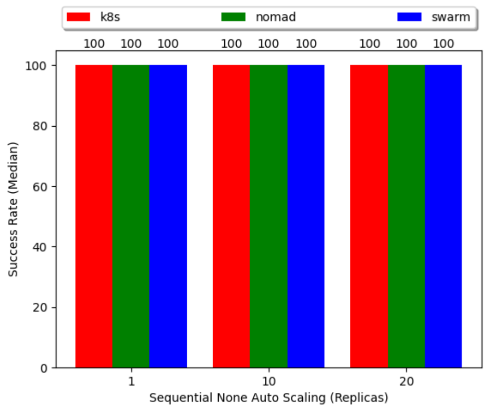
(c) Throughput

Figure 5.14: NodeJS Parallel Auto Scaling Disabled

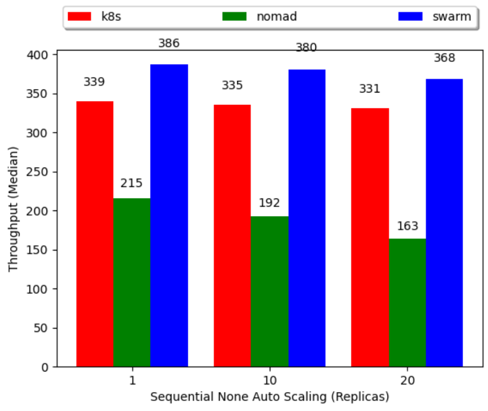
Figure 5.14 shows the results of parallel workloads with auto scaling disabled where the number of concurrency level set to 15 and different number of replicas used (1, 10, 20). Docker Swarm performed better than others in terms of response time and throughput where it achieved the highest throughput (1533 request/second) on replica 10 and lowest response time (8ms) at both replica (10,20). Success rate was identical for all container orchestrators. Increasing the number of replicas had positive impact on response time and throughput in all container orchestrators except Nomad where response time did not change among all replicas and throughput did not show so much enhancement. The explanations for Nomad results could be because that Nomad provider was not able to initiate the required number of replicas inside the Nomad cluster to serve all requests.



(a) Response Time



(b) Success Rate



(c) Throughput

Figure 5.15: NodeJS Sequential Auto Scaling Disabled

Figure 5.15 shows the results of sequential workloads with auto scaling disabled where different number of replicas used (1, 10, 20). Docker swarm performed slightly better than others in terms of response time and throughput. The range values of response time results were very close to each other whereas throughput range values varied and the preferability was for the Docker Swarm where the highest achieved throughput was (386 requests/second) on replica 1 and the lowest response time (2ms) was on all replicas. On that other hand, the success rate for all container orchestrators was identical. Based on the result presented, increasing number of replicas had no impact on the the defined metrics as it appeared that only one replica was utilized which explain those results.

	nomad	**swarm**
k8s	▲ ▲ ▲ ▲	▽ ▽ ▽ ▽
nomad		▽ ▽ ▽ ▽

(a) Response Time

	nomad	**swarm**
k8s	▲ ▲ ▲ ▲	▽ ▽ ▽ ▽
nomad		▽ ▽ ▽ ▽

(b) Throughput

	nomad	**swarm**
k8s	▲ ▲ ▲ –	– – – –
nomad		▽ ▽ ▽ –

(c) Success Rate

Table 5.13: Wilcoxon NodeJS Parallel Auto Scaling Enabled

Table 5.13 represents the *Wilcoxon* results for parallel workloads with auto scaling enabled using four concurrency levels (5, 10, 20, 50) where the results for all metrics were statistically significant for all container orchestrators pairs. The response time results for all pairs were statistically significant with p-values $< 5\%$ which aligned with our previous results where Docker Swarm and K8S performed better than Nomad. Moreover, the throughput results were statistically significant between all pairs with p-values $< 5\%$ and the results showed how Docker Swarm performed better than K8S & Nomad. Finally, the success rate results were statistically significant for (K8S, Nomad) & (Nomad, Docker Swarm) pairs on all concurrency levels except on level 50 and it's noticeable that the results of (K8S, Docker Swarm) were not statistically significant as they had the same success rate in all concurrency levels. All the previous statistical tests aligned with our previous results about the behaviour of NodeJs function deployed across different container orchestrators.

Table 5.14 represents the *Wilcoxon* results for parallel workloads with auto scaling disabled using three different replica settings (1, 10, 20) where all the collected results were statistically significant for response time and throughput of all container orchestrators pairs. The results of response time were statistically significant for all replica settings between all container orchestrators pairs with p-values $< 5\%$. The

	nomad	swarm
k8s	▲ ▲ ▲	▽ ▽ ▽
nomad		▽ ▽ ▽

(a) Response Time

	nomad	swarm
k8s	▲ ▲ ▲	▽ ▽ ▽
nomad		▽ ▽ ▽

(b) Throughput

	nomad	swarm
k8s	− − −	− − −
nomad		− − −

(c) Success Rate

Table 5.14: Wilcoxon NodeJS Parallel Auto Scaling Disabled

same thing also applied to the results obtained for throughput where the direction and color of triangles showed that all p-values $< 5\%$ and how the Docker Swarm performed better than both Nomad and K8S and how K8S also performed better than Nomad. Finally, based on the results provided from the *Wilcoxon* tests, the results were not statistically significant between all container orchestrators pairs as all of them had the same success rate

	nomad	swarm
k8s	▲ ▲ ▲	▽ ▽ ▽
nomad		▽ ▽ ▽

(a) Response Time

	nomad	swarm
k8s	▲ ▲ ▲	▽ ▽ ▽
nomad		▽ ▽ ▽

(b) Throughput

	nomad	swarm
k8s	− − −	− − −
nomad		− − −

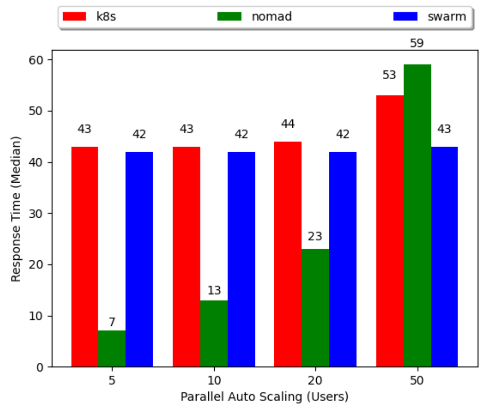
(c) Success Rate

Table 5.15: Wilcoxon NodeJS Sequential Auto Scaling Disabled

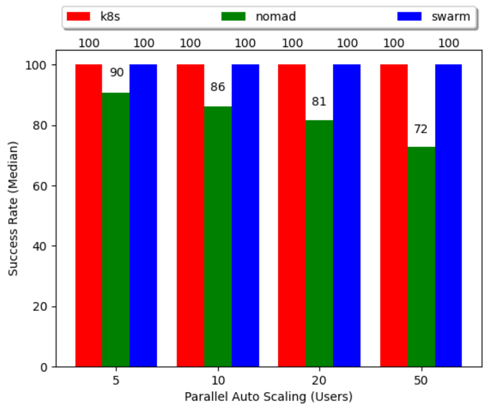
Table 5.15 represent the *Wilcoxon* results for sequential workloads with auto scaling disabled using the same replica settings used before where the results of response time and throughput were statistically significant for all container orchestrators pairs with p-values $< 5\%$. The statistical tests obtained for sequential workloads were similar to the results of parallel workloads with auto scaling disabled for all metrics results across all replica settings.

5.2.3 Java Function

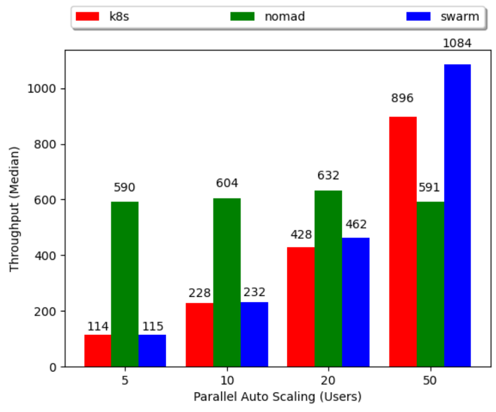
This scenario represents a Serverless function was written in Java 8 that returns simple message on call invocation. This function used a complied programming language where previous two functions were used interpreted languages. It covered both parallel and sequential test cases as illustrated on Figures 5.16, 5.17 and 5.18.



(a) Response Time



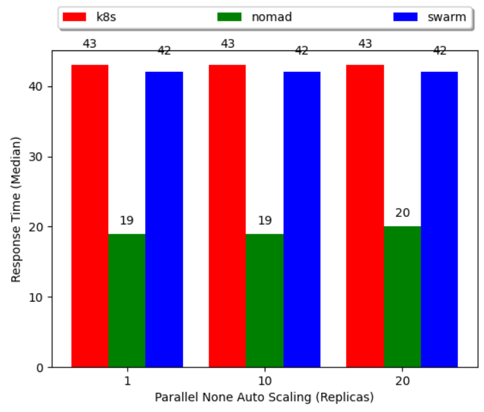
(b) Success Rate



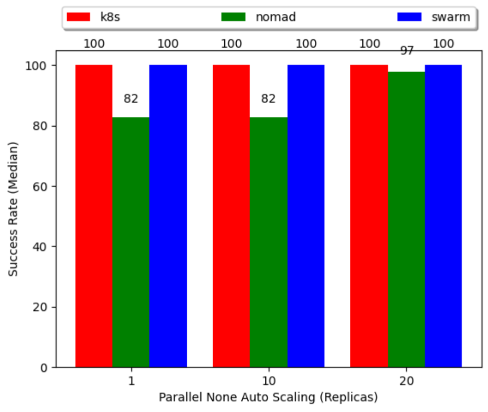
(c) Throughput

Figure 5.16: Java Parallel Auto Scaling Enabled

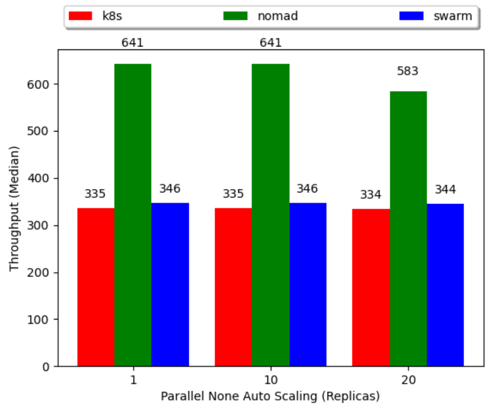
Figure 5.16 shows the results for parallel workloads with auto scaling enabled. Nomad performed better than others in terms of response time and throughput in all concurrency levels except on level 50 where Docker Swarm and K8S were better. Success rate was identical for both Docker Swarm and K8S whereas the Nomad has error rate varied form 10% - 28%. Docker Swarm only had a better results of response time and throughput on concurrency level 50 where it achieved (1084 requests/second) and (43 ms) whereas Nomad achieved the highest throughput (632) on concurrency level 20 and the lowest response time (7ms) on concurrency level 5.



(a) Response Time



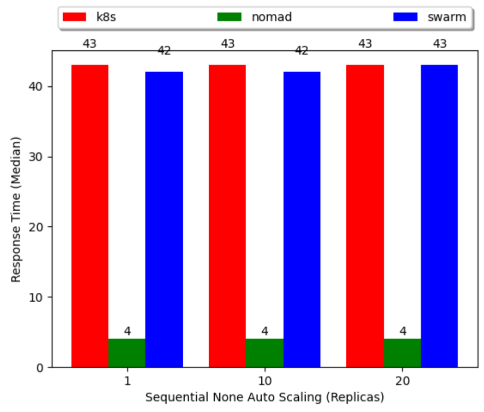
(b) Success Rate



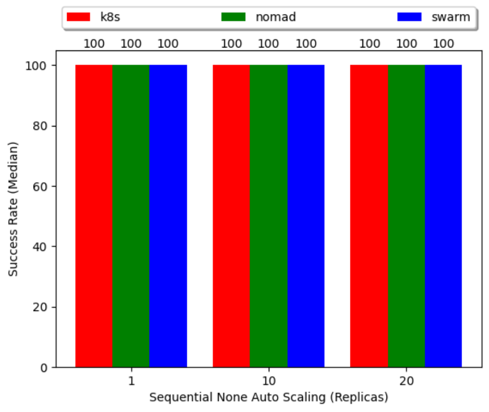
(c) Throughput

Figure 5.17: Java Parallel Auto Scaling Disabled

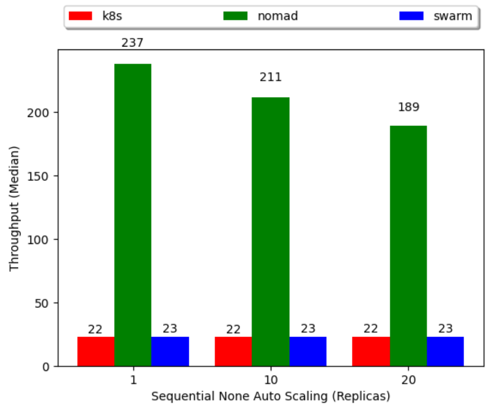
Figure 5.17 represents the results of parallel workloads with auto scaling disabled where fixed number of concurrency level set to 15 and different number of replicas was used (1, 10, 20). Nomad had the best results in terms of response time and throughput using all replicas where it achieved the highest throughput (641 requests/seconds) on replica 10 and the lowest response time (19ms) at both replica (1, 10). However, success rate for both Docker Swarm and K8S was the same with 100%, whereas Nomad had some error rate varied from 3% - 18%. It was observable that for all container orchestrators the increasing on replicas had no clear impact on the response and throughput results.



(a) Response Time



(b) Success Rate



(c) Throughput

Figure 5.18: Java Sequential Auto Scaling Disabled

Figure 5.18 shows the results of sequential workloads with auto scaling disabled that used different replicas settings (1, 10, 20). Nomad performed better than others in terms of response time and throughput in all replica settings where it achieved the highest throughput (237 requests/second) on replica 1 and the lowest response time (4ms) in all replicas . Success rate had identical results for all container orchestrators with 100% success rate. Nomad had 9X throughput than K8S and Docker Swarm and 10X response time enchantment.It was noticeable that increasing the replicas had no clear impacts on the results for all container orchestrators because they were not fully utilized and only one replica could be enough to handle all requests.

	nomad	**swarm**
k8s	▽ ▽ ▽ –	▽ ▽ ▽ ▽
nomad		▲ ▲ ▲ ▽

(a) Response Time

	nomad	**swarm**
k8s	▽ ▽ ▽ ▲	▽ ▽ ▽ ▽
nomad		▲ ▲ ▲ ▽

(b) Throughput

	nomad	**swarm**
k8s	▲ ▲ ▲ ▲	– – – –
nomad		▽ ▽ ▽ ▽

(c) Success Rate

Table 5.16: Wilcoxon Java Parallel Auto Scaling Enabled

Table 5.16 represents the *Wilcoxon* results for parallel workloads with auto scaling enabled using four concurrency levels (5, 10, 20, 50) where the response time, throughput and success rate results were statistically significant for all container orchestrators pairs with some exceptions. The response time results were statistically significant for all pairs using all concurrency levels except for level 50 of (K8S, Nomad) pair. Based on the statistical tests results for response time, the (K8S, Nomad) pair p-values $<$ 5% with preferability to Nomad on all concurrency levels except on level 50. Moreover, the results of (K8S, Docker Swarm) p-values $<$ 5% where Docker Swarm performed better than K8S and the p-values results of (Nomad, Docker Swarm) pair $<$ 5% where Nomad performed better on (5, 10, 20) based on the color and direction of triangles. On the other hand, the throughput results were statistically significant for all pairs on all concurrency levels except on level 50 for (K8S, Nomad) & (Nomad, Docker Swarm). Finally, the success rate results were statistically significant for (K8S, Nomad) & (Nomad, Docker Swarm) pairs where p-values $<$ 5% for all concurrency levels and the results aligned with our previous results for success rate results.

Table 5.17 represents the *Wilcoxon* results for parallel workloads with auto scaling disabled using three different replica settings where all the results of the metrics were statistically significant for all container orchestrators pairs except for success rate results of (K8S, Docker Swarm) pair. The results of response time and throughput were statistically significant with p-values $<$ 5% for all replica settings and the preferability went to Nomad over K8S & Docker Swarm which aligned with our

	nomad	**swarm**
k8s	▽ ▽ ▽	▽ ▽ ▽
nomad		▲ ▲ ▲

(a) Response Time

	nomad	**swarm**
k8s	▽ ▽ ▽	▽ ▽ ▽
nomad		▲ ▲ ▲

(b) Throughput

	nomad	**swarm**
k8s	▲ ▲ ▲	– – –
nomad		▽ ▽ ▽

(c) Success Rate

Table 5.17: Wilcoxon Java Parallel Auto Scaling Disabled

previous results. Finally, the success rate results were statistically significant for (K8, Nomad) & (Nomad, Docker Swarm) because of the error rate generated on the Nomad side which explains the statistical test results.

	nomad	**swarm**
k8s	▽ ▽ ▽	▽ ▽ –
nomad		▲ ▲ ▲

(a) Response Time

	nomad	**swarm**
k8s	▽ ▽ ▽	▽ ▽ ▽
nomad		▲ ▲ ▲

(b) Throughput

	nomad	**swarm**
k8s	– – –	– – –
nomad		– – –

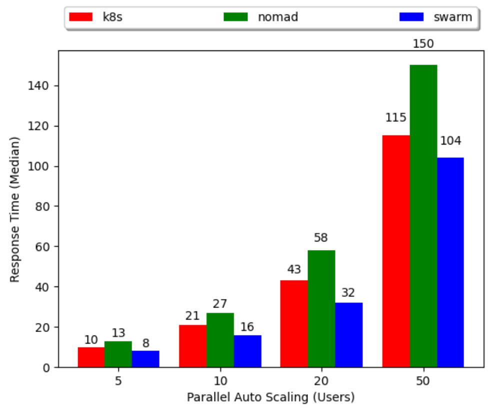
(c) Success Rate

Table 5.18: Wilcoxon Java Sequential Auto Scaling Disabled

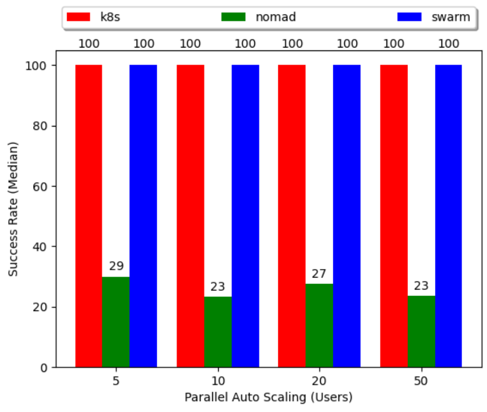
Table 5.18 represents the *Wilcoxon* results for sequential workloads with auto scaling disabled using the same replica settings used before where both response time and throughput results are statistically significant for all container orchestrators pairs using all replicas except on (K8S, Docker Swarm) pair using replica 20. The response time and throughput statistical are nearly similar to the results we obtained for parallel test cases with auto scaling disabled where p-values $< 5\%$ where preferability of results are for Nomad based on the direction and color of the triangles. However, the results for success rate are not statistically significant for any of the defined pairs as all of the container orchestrators have the same success rate across all runs using all defined replica settings.

5.2.4 Go Function

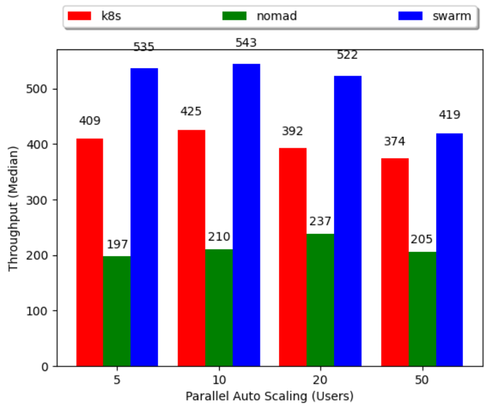
This scenario represents a Serverles function written in Go 1.13 that returns a simple message on call invocation. Go is categorized as complied programming language like Java. Figures 5.19, 5.20 and 5.21 present results of parallel and sequential test cases.



(a) Response Time



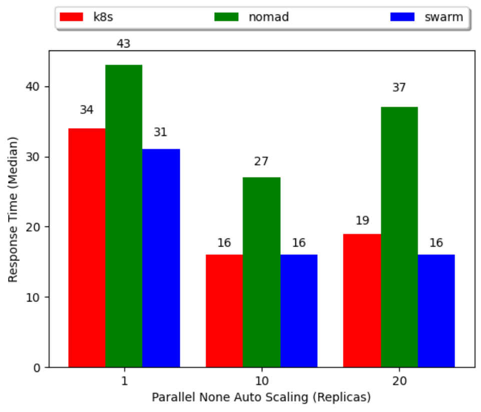
(b) Success Rate



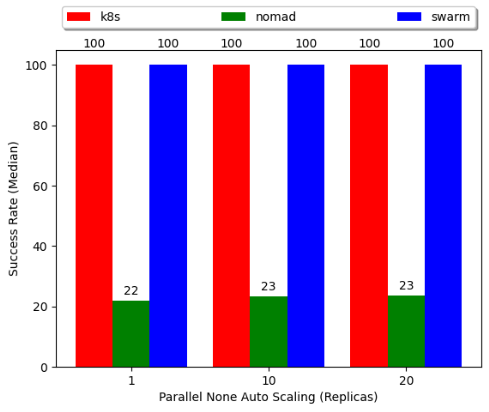
(c) Throughput

Figure 5.19: Go Parallel Auto Scaling Enabled

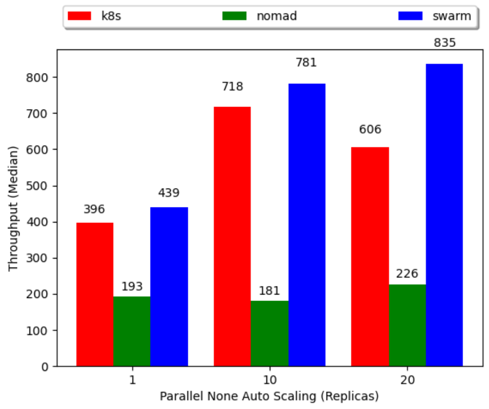
Figure 5.19 represents the results of parallel workloads with auto scaling enabled. Docker Swarm performed better than others in term of response time and throughput. Success rate was identical for both Docker Swarm and K8S. However, the success rate had the worst result for Nomad where error rate was > 76%. Docker Swarm achieved the highest throughput between others on concurrency level 10 with (543 requests/seconds) where it had 2X performance than Nomad and nearly 1.5X than K8S. Moreover, it achieved the lowest response time (8ms) on concurrency level 5.



(a) Response Time



(b) Success Rate

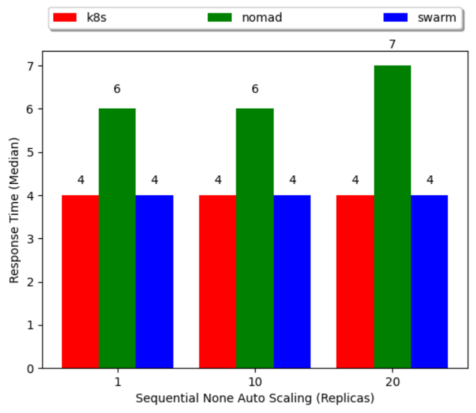


(c) Throughput

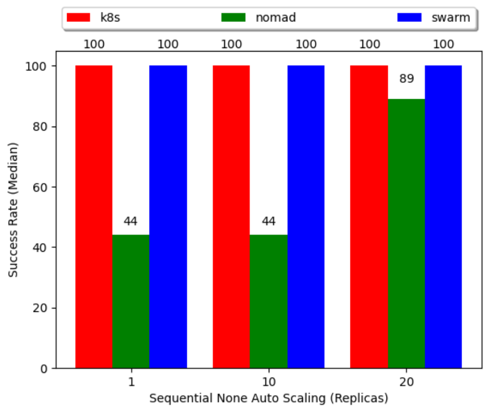
Figure 5.20: Go Parallel Auto Scaling Disabled

Figure 5.20 represents the results of parallel workloads with auto scaling disabled where fixed concurrency level set to 15 and different number of replicas used (1, 20, 20). Docker Swarm performed better than others in terms of response time and throughput in all test cases. However, success rate results were identical for both Docker Swarm and K8S whereas Nomad had the worst success rate results where error rate > 76%. Increasing the number of replicas had noticeable impact for both

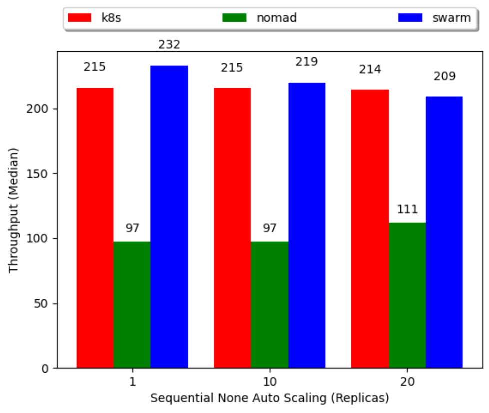
Docker Swarm and K8S in terms of response time and throughput and small impact for Nomad. Docker Swarm achieved the highest throughput on replica 20 with (835 requests/second) and lowest response time (16ms) on replica 20 as well.



(a) Response Time



(b) Success Rate



(c) Throughput

Figure 5.21: Go Sequential Auto Scaling Disabled

Figure 5.21 represents the results of sequential workloads with auto scaling disabled where one request at a time was sent and different number of replicas was used (1, 10, 20). Docker Swarm and K8S had identical results for response time and success rate. However, Nomad had worst result in terms of all metrics, especially for success rate where the error rate was high and varied from 11% - 56%. On the other hand, Docker Swarm performed better than K8S using replicas 1 &10 and K8S had the preferability on replica 20. Generally, Increasing replicas for sequential workloads had small impact on throughput and no impact on response time for all container orchestrators. The reason of the high error rate for Nomad was because of Bad Gateway (502) responses where the OpenFaas gateway was unable to get the response back from the created function (container).

	nomad	**swarm**
k8s	▲ ▲ ▲ ▲	▽ ▽ ▽ ▽
nomad		▽ ▽ ▽ ▽

(a) Response Time

	nomad	**swarm**
k8s	▲ ▲ ▲ ▲	▽ ▽ ▽ ▽
nomad		▽ ▽ ▽ ▽

(b) Throughput

	nomad	**swarm**
k8s	▲ ▲ ▲ ▲	– – – –
nomad		▽ ▽ ▽ ▽

(c) Success Rate

Table 5.19: Wilcoxon Go Parallel Auto Scaling Enabled

Table 5.19 represents the *Wilcoxon* results for parallel workloads with auto scaling enabled using four concurrency levels (5, 10, 20, 50) where the results of response time, throughput and success rate were statistically significant for all container orchestrators pairs on all concurrency levels with an exception for success rate result of (K8S, Docker Swarm) where they had the same success rate results which explains why the statistical tests were not significant. It's clear that both response time and throughput for container orchestrators pairs under all concurrency levels were statistically significant where p-values $< 5\%$. Moreover, the direction and color of triangles showed how K8S & Docker Swarm performed better than Nomad which aligned with our previous results. Finally, the success results were statistically significant for (K8S, Nomad) & (Nomad, Docker Swarm) pairs where p-values $< 5\%$ of all replica settings because of the high error rates on Nomad.

Table 5.20 represents the *Wilcoxon* results for parallel workloads with auto scaling disabled using three different replica settings (1, 10, 20). The results of response time, throughput and success rate were statistically significant and aligned with our previous results. The results of response time and throughput for all container orchestrators pairs were statistically significant with p-values $< 5\%$ except on replica 10 of (K8S, Docker Swarm). Based on the direction and color of triangles, the results showed how both Docker Swarm and K8S performed better than Nomad which aligned with our previous results. Finally, the results of success rate were statistically significant for (K8S, Nomad) & (Nomad, Docker Swarm) pairs because

	nomad	**swarm**
k8s	▲ ▲ ▲	▽ – ▽
nomad		▽ ▽ ▽

(a) Response Time

	nomad	**swarm**
k8s	▲ ▲ ▲	▽ – ▽
nomad		▽ ▽ ▽

(b) Throughput

	nomad	**swarm**
k8s	▲ ▲ ▲	– – –
nomad		▽ ▽ ▽

(c) Success Rate

Table 5.20: Wilcoxon Go Parallel Auto Scaling Disabled

of the high error rates on Nomad which explains the result of the p-values $< 5\%$.

	nomad	**swarm**
k8s	▲ ▲ ▲	– – –
nomad		▽ ▽ ▽

(a) Response Time

	nomad	**swarm**
k8s	▲ ▲ ▲	▽ – –
nomad		▽ ▽ ▽

(b) Throughput

	nomad	**swarm**
k8s	▲ ▲ ▲	– – –
nomad		▽ ▽ –

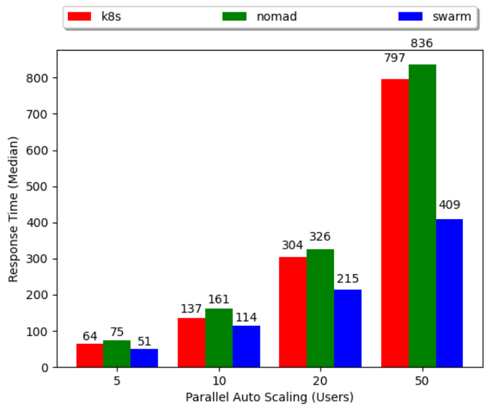
(c) Success Rate

Table 5.21: Wilcoxon Go Sequential Auto Scaling Disabled

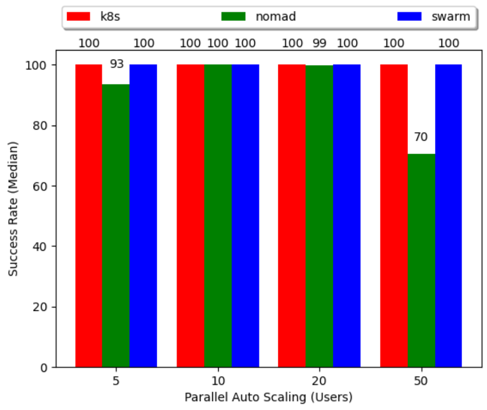
Table 5.21 represents the *Wilcoxon* results for sequential workloads with auto scaling disabled using the same replica settings used before. The results for response time, throughput and success rate were statistically significant mainly for (K8S, Nomad) & (Nomad, Docker Swarm) pairs where the relationship between Docker Swarm and K8S were not statistically significant which means that running go function either on Docker Swarm or K8S under sequential workloads for all replica settings will not make a difference except for throughput on replica 1 of (K8S, Docker Swarm) pair where p-value is $< 5\%$.

5.3 Chaining Serverless Functions Scenario

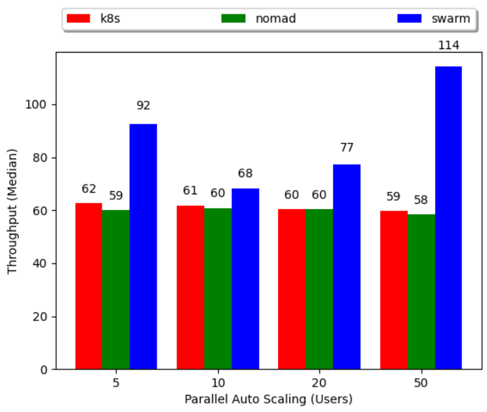
This section represents the results of invoking chaining Serverless functions where source function invokes destination function that accepts the dimension of matrix to do a matrix multiplication. For example if source function pass parameter = 20, then the destination function will initialize two matrices with 20 X 20 by applying matrix multiplication and return the result. Figures 5.22, 5.23 and 5.24 represent both parallel and sequential test cases.



(a) Response Time

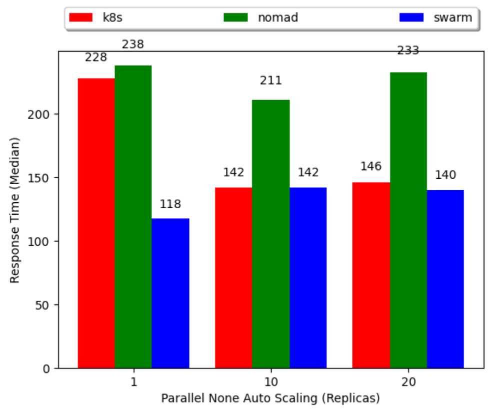


(b) Success Rate

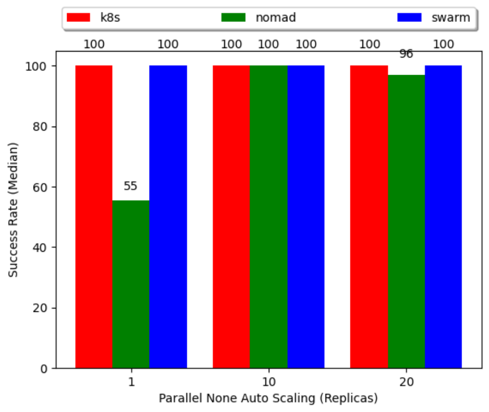


(c) Throughput

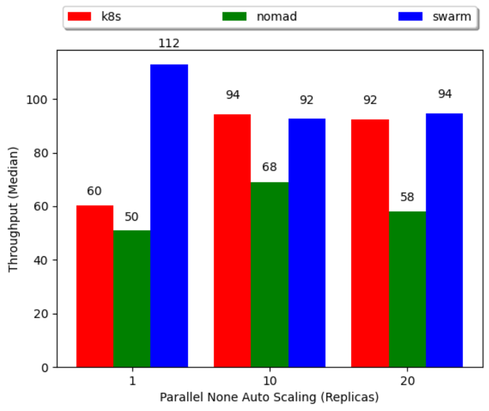
Figure 5.22: Serverless Chaining Parallel Auto Scaling Enabled



(a) Response Time

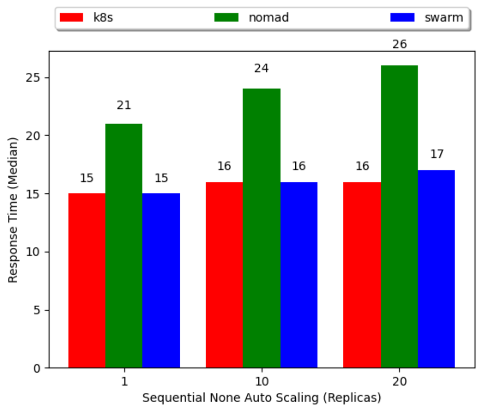


(b) Success Rate

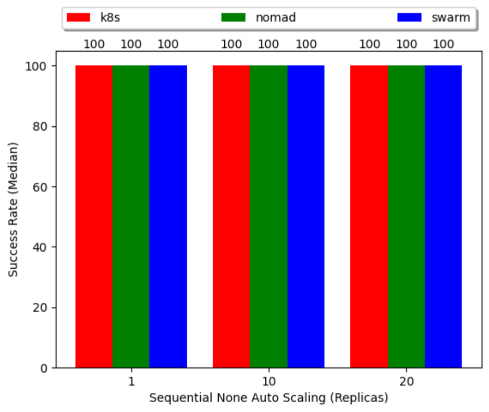


(c) Throughput

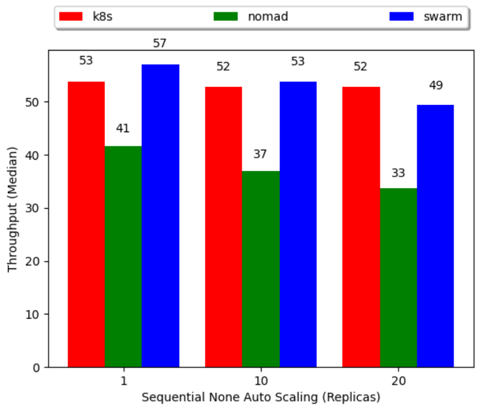
Figure 5.23: Serverless Chaining Parallel Auto Scaling Disabled



(a) Response Time



(b) Success Rate



(c) Throughput

Figure 5.24: Serverless Chaining Sequential Auto Scaling Disabled

Figure 5.22 shows the results of parallel workloads with auto scaling enabled. Docker Swarm performed better than other in terms of response time and throughput. The success rate had identical results for Docker Swarm and K8S whereas Nomad had error rate varied from 7% - 30% on concurrency level 5 and 50. On the other hand, Docker Swarm achieved the highest value on concurrency level 50 (114 requests/second) and had best results over others in all concurrency levels. Moreover, Nomad and K8S nearly had the same throughput results.

Figure 5.23 shows the results for parallel workloads with auto scaling disabled where fixed concurrency level 15 and different settings of replicas (1, 10, 20) were used. Docker Swarm performed better than others in terms of response time and throughput for all replicas except on replica 10 where K8S had better result for throughput recorded as (94 requests/second) comparing to (92 requests/second). Moreover, success rate had identical result for both Docker Swarm and K8S whereas Nomad had error rate varied from 4% - 45%.

Figure 5.24 shows the results of sequential workloads with auto scaling disabled where 15 concurrency level and different settings of replicas (1, 10, 20) were used. Docker Swarm and K8S had results close to each other in terms of response time and throughput with very small variations in results whereas Nomad had worst results among them for response time and throughput. Moreover, success rate had identical results for all container orchestrators. Its noticeable that the range of throughput result values were low for all container orchestrators and the reason was the complexity of chaining functions, implementation logic and the sequential workloads where only one request at a time was sent. Finally, the increasing on replicas for sequential workloads had no observable impact on the response time and throughput.

	nomad	**swarm**
k8s	▲ ▲ ▲ –	▽ ▽ ▽ ▽
nomad		▽ ▽ ▽ ▽

(a) Response Time

	nomad	**swarm**
k8s	▲ ▲ – ▲	▽ ▽ ▽ ▽
nomad		▽ ▽ ▽ ▽

(b) Throughput

	nomad	**swarm**
k8s	▲ – – ▲	– – – –
nomad		▽ – – ▽

(c) Success Rate

Table 5.22: Wilcoxon Serverless Chaining Parallel Auto Scaling Enabled

The table 5.22 represents the *Wilcoxon* results for parallel workloads with auto scaling enabled using four concurrency levels (5, 10, 20, 50). The results were statistically significant for all metrics. The results for response time for all container orchestrators pairs were statistically significant with p-values $< 5\%$ for all concurrency levels except on level 50 for (K8S, nomad) pair. On the other hand, the results (K8S, Docker Swarm) & (Nomad, Docker Swarm) pairs were statistically significant where Docker Swarm had the preferability among other orchestrators. The same thing applied for throughput results where all container orchestrators pairs were statistically significant with p-values $< 5\%$ for all concurrency levels except on level 20 for (K8S, nomad) pair. Finally, the success rate results were statistically significant on concurrency levels (10, 50) for (K8S, Nomad) & (Nomad, Docker Swarm) pairs.

	nomad	**swarm**
k8s	– ▲ ▲	▽ – –
nomad		▽ ▽ ▽

(a) Response Time

	nomad	**swarm**
k8s	▲ ▲ ▲	▽ ▲ –
nomad		▽ ▽ ▽

(b) Throughput

	nomad	**swarm**
k8s	▲ – ▲	– – –
nomad		▽ – ▽

(c) Success Rate

Table 5.23: Wilcoxon Serverless Chaining Parallel Auto Scaling Disabled

Table 5.23 represents the *Wilcoxon* results for parallel workloads with auto scaling disabled using three different replica settings (1, 10, 20). The results were statistically significant for all container orchestrators pairs at least for one replica in all metrics except for success rate where only two pairs were statistically significant with p-values $< 5\%$ (K8S, Nomad) & (Nomad, Docker Swarm). The response time results were statistically significant on all replica settings for (Nomad, Docker Swarm) where Docker Swarm achieved better results. Moreover, the results of (K8S, Nomad) & (K8S, Docker Swarm) pairs were also statistically significant on replica settings (10, 20), (1) respectively. On the other hand, the throughput results for (K8S, Nomad) & (Nomad, Docker Swarm) were statistically significant on all replica settings. However, the throughput results for (K8S, Docker Swarm) were statistically significant only on replica settings (1, 10). All the statistical tests aligned with the results we obtained before for chaining function test cases. Finally, the success rate results were only statistically significant for (K8S, Nomad) & (Nomad, Docker Swarm) on replica settings (1, 20).

	nomad	**swarm**
k8s	▲ ▲ ▲	– – ▲
nomad		▽ ▽ ▽

(a) Response Time

	nomad	**swarm**
k8s	▲ ▲ ▲	▽ – ▲
nomad		▽ ▽ ▽

(b) Throughput

	nomad	**swarm**
k8s	– – –	– – –
nomad		– – –

(c) Success Rate

Table 5.24: Wilcoxon Serverless Chaining Sequential Auto Scaling Disabled

Table 5.24 represents the *Wilcoxon* results for sequential workloads using the same replica settings in previous test cases. The generated statistical tests showed that all container orchestrators pairs were statistically significant for response time and throughput at least using one of the replica settling defined where p-values $< 5\%$. The response time and throughput for (K8S, Nomad) & (Nomad, Docker Swarm) pairs were statistically significant using all replica settings. However, the response time and throughput for (K8S, Docker Swarm) pair were statistically significant on replica settings 20, (1, 20) respectively. Finally, the success rate for all container orchestrators pairs were not statistically significant using all replica settings as all the orchestrators share the same success rate across all result runs.

5.4 Warm and Cold Start Scenario

This section represents the results of enforcing cold start for Serverless functions by configuring the inactivity duration provided by OpenFaas framework where it was set to 6 minutes which means if no request hit the function within this period it will be scaled automatically to zero replicas instead of keeping the function alive without doing anything. On the other hand, warm start results were also collected and compared with cold start results. Conducting this scenario was different from others where invocation of total requests was divided into 6 intervals (chunks) that simulated warm and cold start scenarios as shown in Figure 5.25 where the results of 3 pairs (warm —> wait —> cold) were generated. The warm & cold start scenario addresses H_{13}-H_{15} hypotheses.

Based on Figure 5.25 between each interval (chunk) there was an idle inactivity duration for 10 minutes which was larger than the configured inactivity for OpenFaas

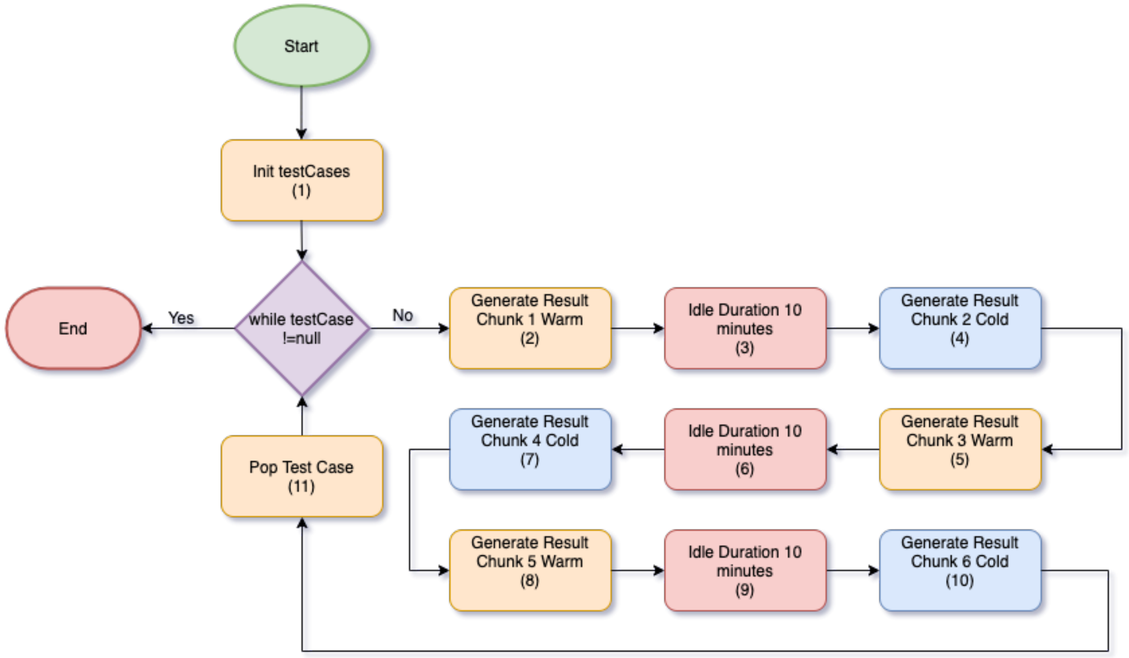
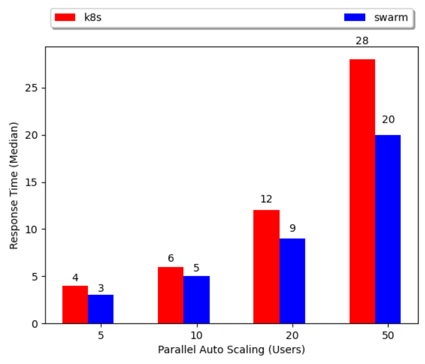

Start
Init testCases (1)
while testCase !=null
Yes
End
No
Generate Result Chunk 1 Warm (2)
Idle Duration 10 minutes (3)
Generate Result Chunk 2 Cold (4)
Generate Result Chunk 3 Warm (5)
Idle Duration 10 minutes (6)
Generate Result Chunk 4 Cold (7)
Generate Result Chunk 5 Warm (8)
Idle Duration 10 minutes (9)
Generate Result Chunk 6 Cold (10)
Pop Test Case (11)

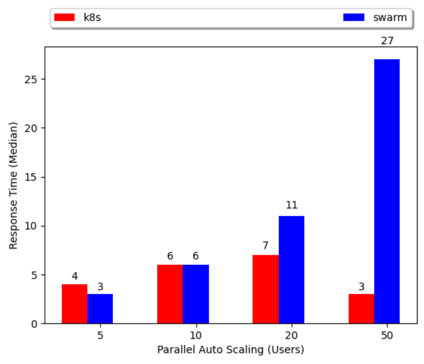
Figure 5.25: Warm & Cold Start Flow

framework. The reason of selection both 6 & 10 minutes was to simplify conducting this scenario as it took 2.5 days to complete it and generate the related results and to make sure that the enforcing of cold start was triggred successfully.

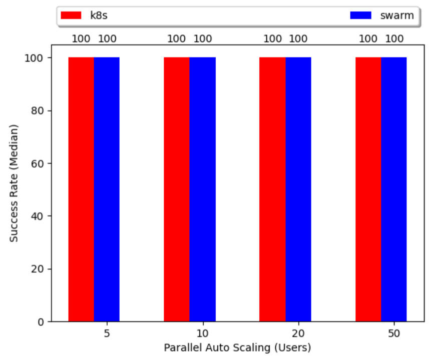
The function implemented for this scenario was written in NodeJS 12 with simple logic that accepts string parameter and return it as simple JSON object. The reason for using this simple implementation was to study the impact of cold/warm start without any other factors that could affect the result. Figures 5.26 - 5.34 represent the results of parallel and sequential test cases for all warm & cold pairs.



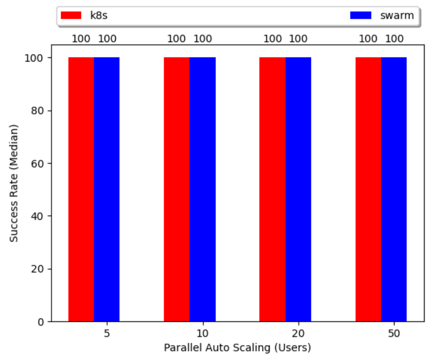
(a) Response Time (Warm)



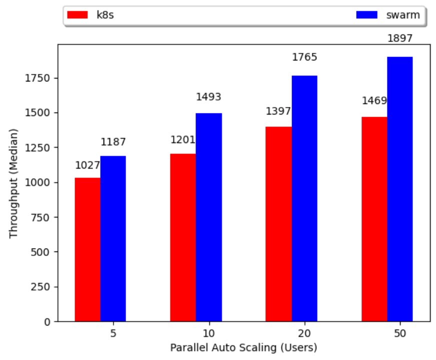
(b) Response Time (Cold)



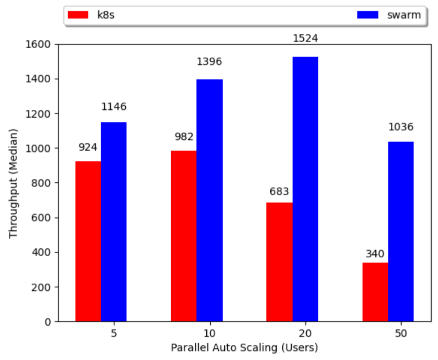
(c) Success Rate (Warm)



(d) Success Rate (Cold)

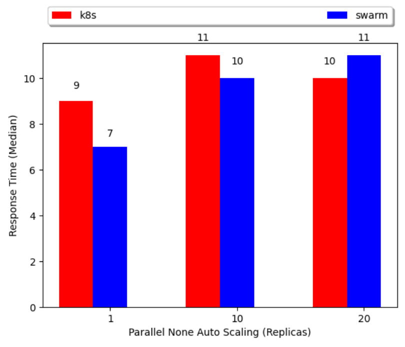


(e) Throughput (Warm)

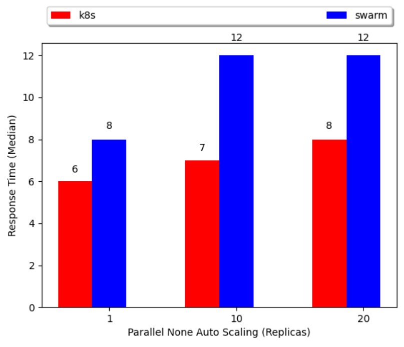


(f) Throughput (Cold)

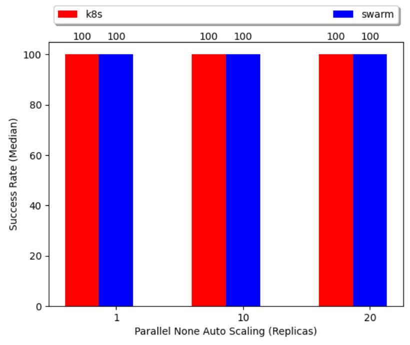
Figure 5.26: First Pair (Warm & Cold) Parallel Auto Scaling Enabled



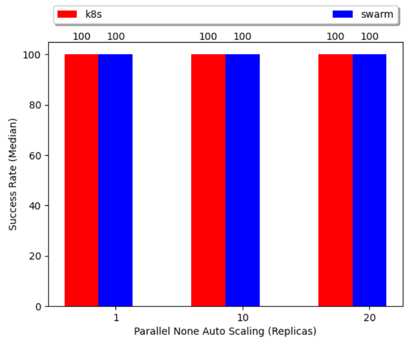
(a) Response Time (Warm)



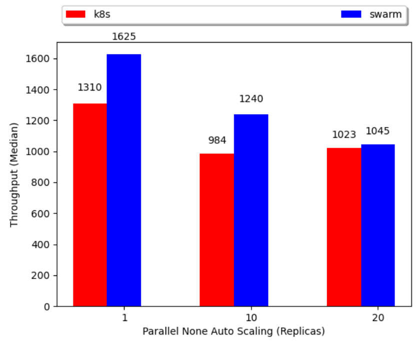
(b) Response Time (Cold)



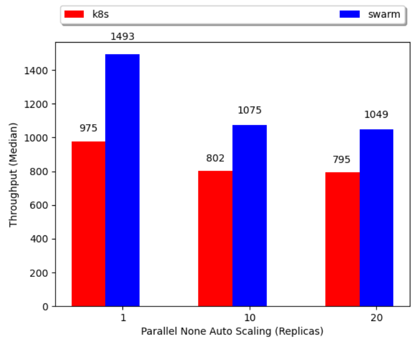
(c) Success Rate (Warm)



(d) Success Rate (Cold)

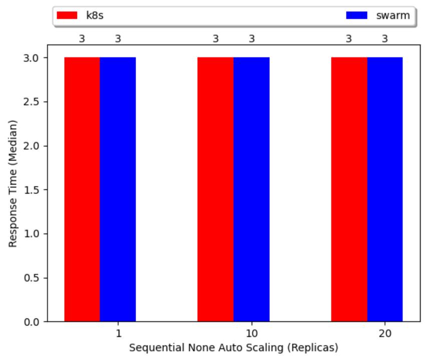


(e) Throughput (Warm)

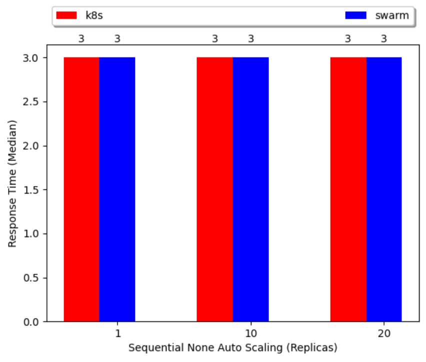


(f) Throughput (Cold)

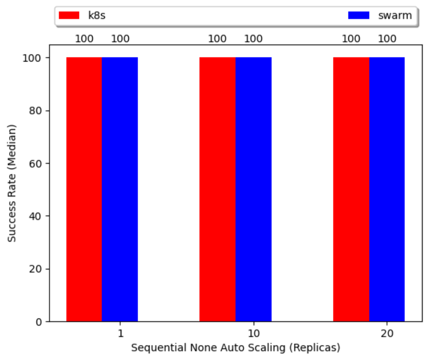
Figure 5.27: First Pair (Warm & Cold) Parallel Auto Scaling Disabled



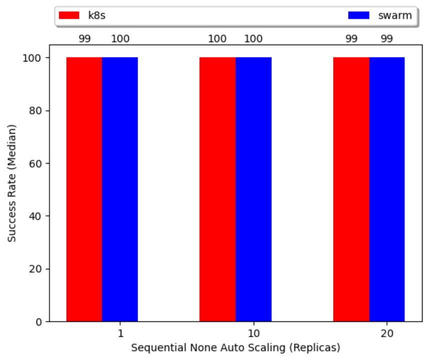
(a) Response Time (Warm)



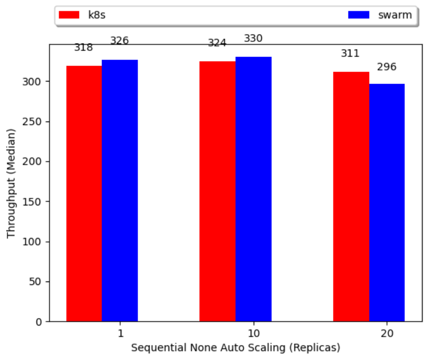
(b) Response Time (Cold)



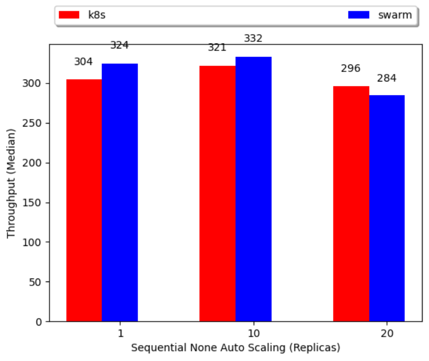
(c) Success Rate (Warm)



(d) Success Rate (Cold)

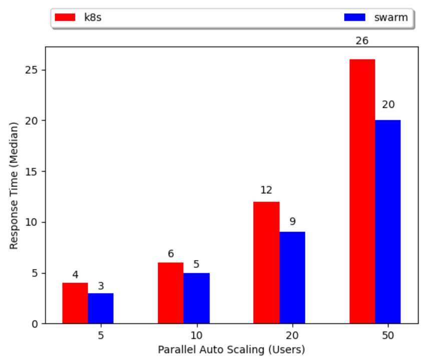


(e) Throughput (Warm)

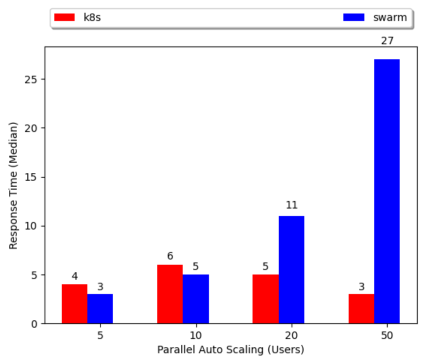


(f) Throughput (Cold)

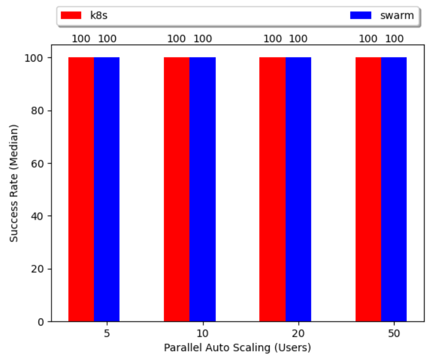
Figure 5.28: First Pair (Warm & Cold) Sequential Auto Scaling Disabled



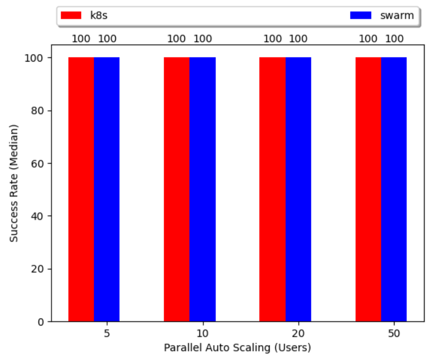
(a) Response Time (Warm)



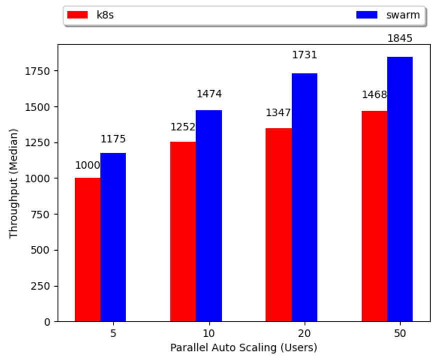
(b) Response Time (Cold)



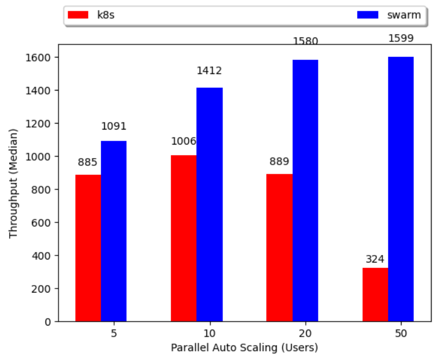
(c) Success Rate (Warm)



(d) Success Rate (Cold)

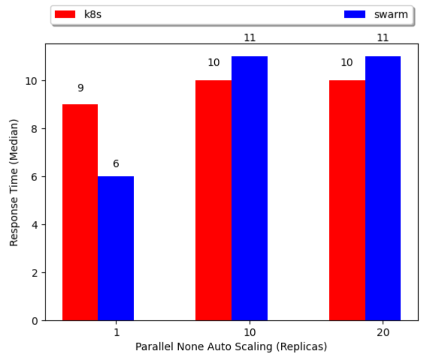


(e) Throughput (Warm)

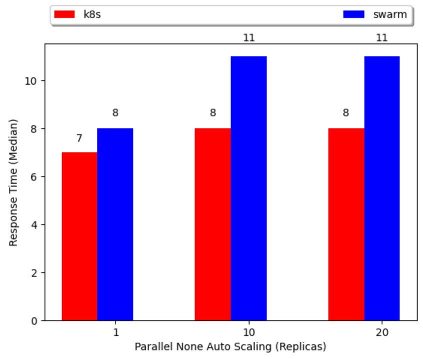


(f) Throughput (Cold)

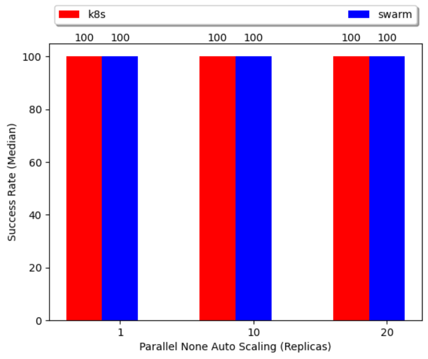
Figure 5.29: Second Pair (Warm & Cold) Parallel Auto Scaling Enabled



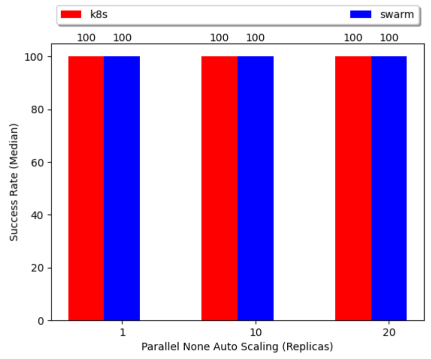
(a) Response Time (Warm)



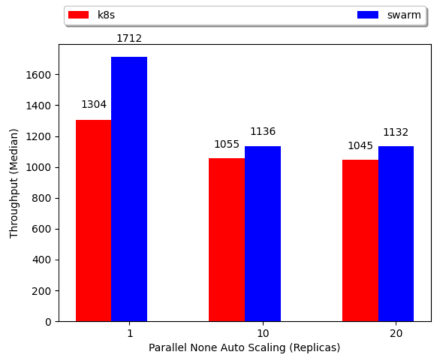
(b) Response Time (Cold)



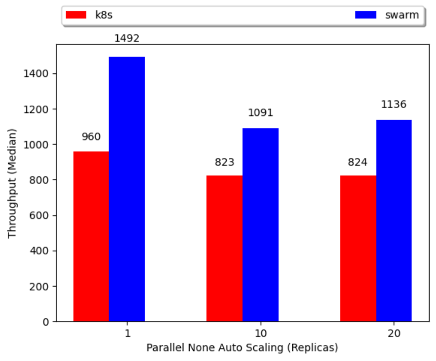
(c) Success Rate (Warm)



(d) Success Rate (Cold)

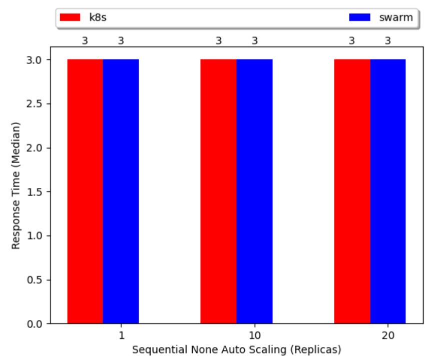


(e) Throughput (Warm)

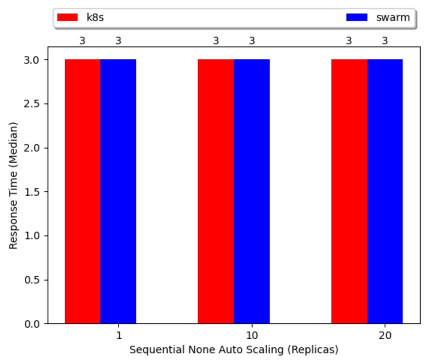


(f) Throughput (Cold)

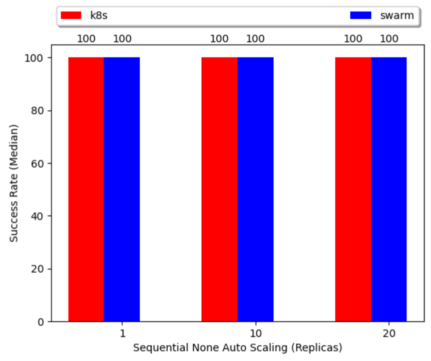
Figure 5.30: Second Pair (Warm & Cold) Parallel Auto Scaling Disabled



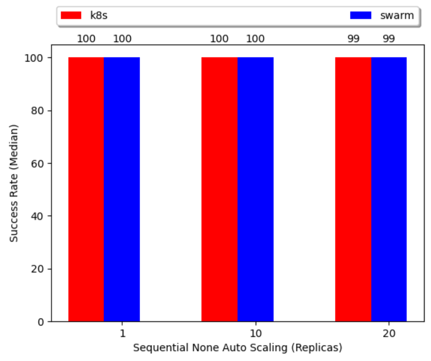
(a) Response Time (Warm)



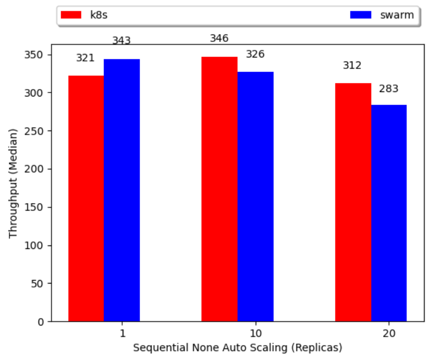
(b) Response Time (Cold)



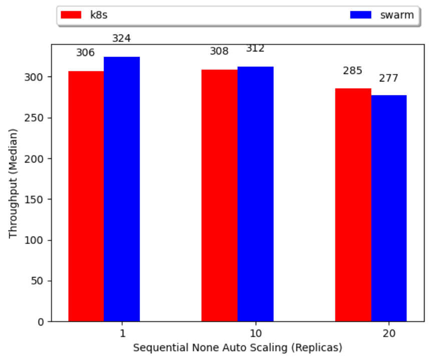
(c) Success Rate (Warm)



(d) Success Rate (Cold)

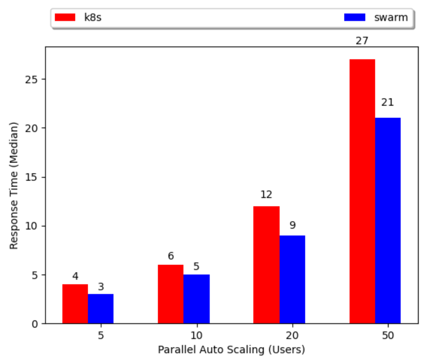


(e) Throughput (Warm)

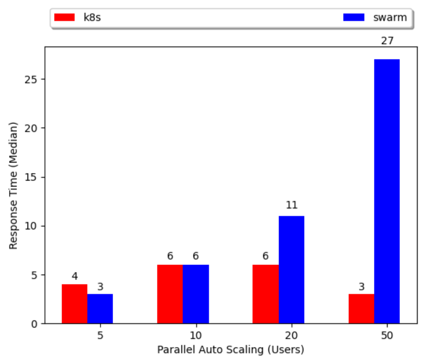


(f) Throughput (Cold)

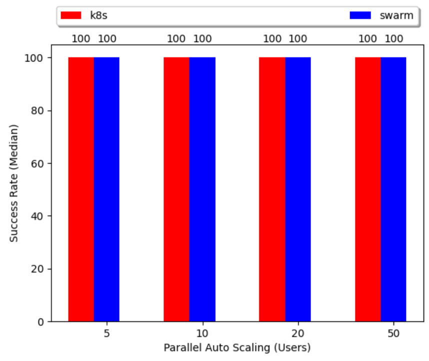
Figure 5.31: Second Pair (Warm & Cold) Sequential Auto Scaling Disabled



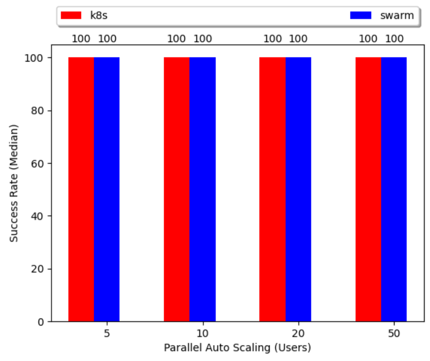
(a) Response Time (Warm)



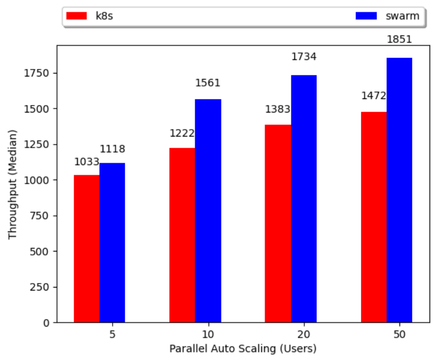
(b) Response Time (Cold)



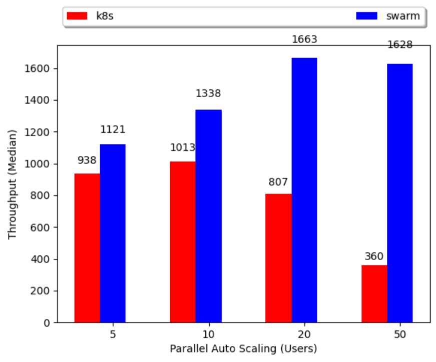
(c) Success Rate (Warm)



(d) Success Rate (Cold)

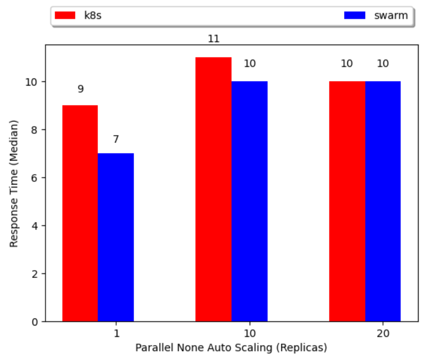


(e) Throughput (Warm)

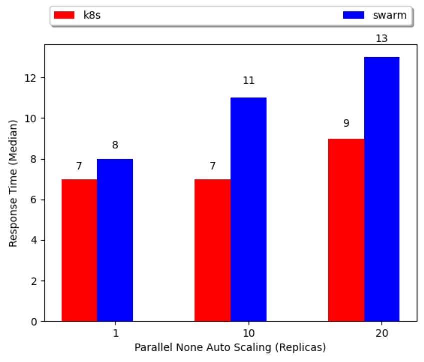


(f) Throughput (Cold)

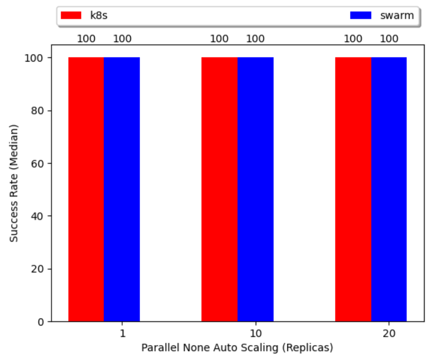
Figure 5.32: Third Pair (Warm & Cold) Parallel Auto Scaling Enabled



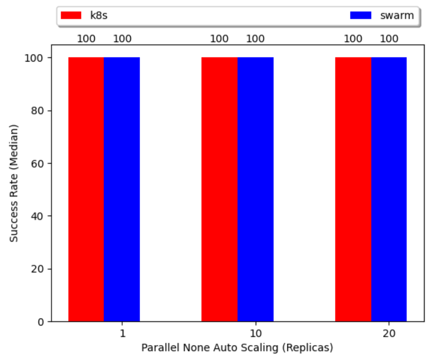
(a) Response Time (Warm)



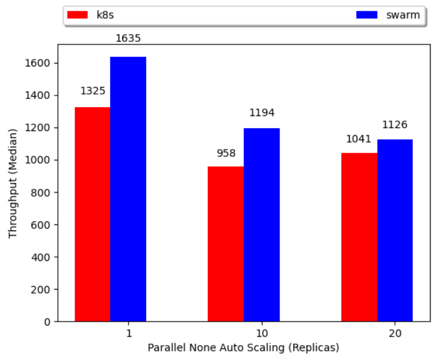
(b) Response Time (Cold)



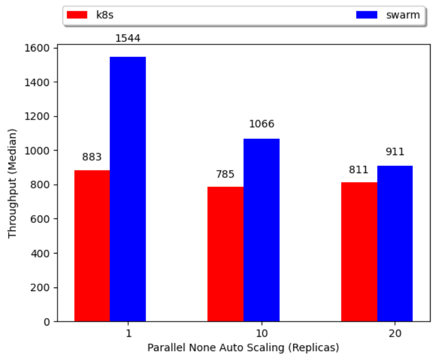
(c) Success Rate (Warm)



(d) Success Rate (Cold)

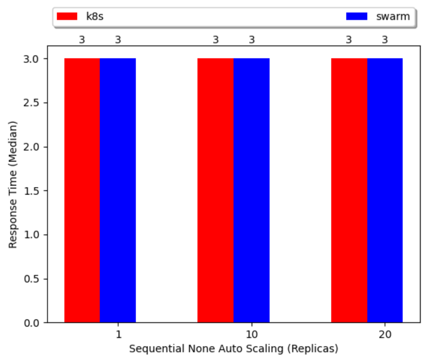


(e) Throughput (Warm)

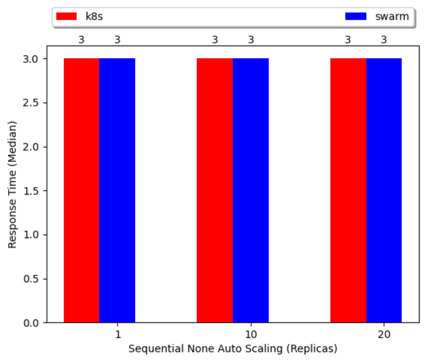


(f) Throughput (Cold)

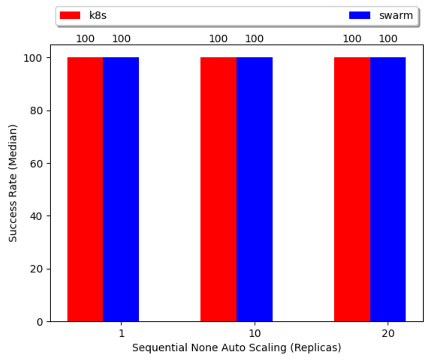
Figure 5.33: Third Pair (Warm & Cold) Parallel Auto Scaling Disabled



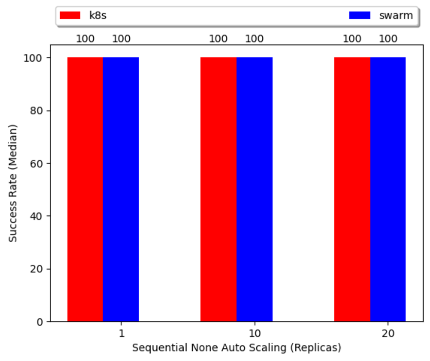
(a) Response Time (Warm)



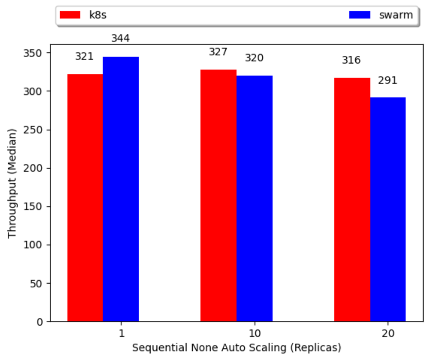
(b) Response Time (Cold)



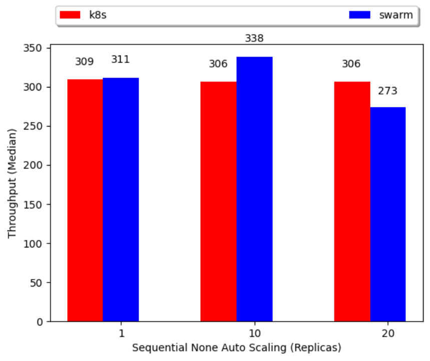
(c) Success Rate (Warm)



(d) Success Rate (Cold)



(e) Throughput (Warm)



(f) Throughput (Cold)

Figure 5.34: Third Pair (Warm & Cold) Sequential Auto Scaling Disabled

Figures 5.26, 5.29 and 5.32 represent the results of parallel workloads with auto scaling enabled generated from all warm & cold pairs. Docker Swarm performed better than K8S in terms of throughput and response time for warm requests in all cases where Docker Swarm achieved the highest throughput (1897 requests/second) with concurrency level 50 and the lowest response time (3ms). Success rate was identical for Docker Swarm and K8S for both warm and cold requests with 100% success rate. However, K8S performed better than Docker Swarm in terms of response time for cold requests where its noticeable in concurrency levels (20, 50) where it achieved the lowest response time (3ms) at concurrency level 50.

Figures 5.27, 5.30 and 5.33 represent the results of parallel workloads with auto scaling disabled generated from all warm and cold pairs. Docker Swarm performed better than K8S in terms of response time and throughput for warm requests nearly in all test cases where Docker Swarm achieved the highest throughput (1712 requests/seconds) and the lowest response time on one replica with (6ms). Success rate had identical results for both Docker Swarm and K8S with 100% success rate in all test cases for both cold & warm requests. However, K8S performed better than Docker Swarm for cold requests in terms of response time in all test cases where it achieved the lowest result (6ms) at replica one.

Figures 5.28, 5.31 and 5.34 represent the results of sequential workloads with auto scaling disabled generated from all warm and cold pairs. Docker Swarm and K8S had identical results for both success rate and response time for all replica settings in warm & cold requests where increasing the number of replicas had no impact on response time and success rate. Comparing the results of throughput for both warm & cold requests, its noticeable that results were close to each other with some preferability to the K8S over Docker Swarm.

Tables 5.25, 5.26 and 5.27 represents the *Wilcoxon* results of all warm and cold pairs for parallel and sequential test cases using different concurrency levels and replica settings between K8S and Docker Swarm. Table 5.25 represents the *Wilcoxon* results of parallel workloads with auto scaling enabled using four concurrency levels (5, 10, 20, 50) for all previous warm and cold pairs results. The results were statistically significant with p-values $< 5\%$ for response time and throughput for both warm and cold test cases using all concurrency levels except for cold scenario of response time using replica 10. The results generated explained how the Docker Swarm performed better than K8S based on the direction of the triangles where most of them had down triangles which means that K8S had worst results than Docker Swarm

	k8s —> swarm
warm	▽ ▽ ▽ ▽
cold	▽ – ▲ ▲

(a) Response Time

	k8s —> swarm
warm	▽ ▽ ▽ ▽
cold	▽ ▽ ▽ ▽

(b) Throughput

	k8s —> swarm
warm	– – – –
cold	– – – –

(c) Success Rate

Table 5.25: Wilcoxon (Warm & Cold) Parallel Auto Scaling Enabled

except on cold test cases for levels (20, 50) where K8S had best results. Finally, the results for success rate were not statistically significant between K8S & Docker Swarm since they had the same success rate across all results.

	k8s —> swarm
warm	▽ – ▲
cold	▲ ▲ ▲

(a) Response Time

	k8s —> swarm
warm	▽ ▽ –
cold	▽ ▽ ▽

(b) Throughput

	k8s —> swarm
warm	– – –
cold	– – –

(c) Success Rate

Table 5.26: Wilcoxon (Warm & Cold) Parallel Auto Scaling Disabled

Table 5.26 represents the *Wilcoxon* results of parallel workloads with auto scaling disabled using three replica settings (1, 10, 20) for both cold and warm test cases. The results were statistically significant with p-values $< 5\%$ for response time and throughput for all replica settings except on warm test case on replica 10 for response time and throughput on replica 20 where K8S & Docker Swarm had the same results. In general, the results for response time showed how K8S performed better than Docker Swarm whereas throughput results showed the preferability of Docker Swarm over K8S. All statistical tests were aligned with our results we obtained before. Finally, the results for success rate were not statistically significant between K8S & Docker Swarm since they had the same success rate across all results.

	k8s —> swarm
warm	− − −
cold	− − −

(a) Response Time

	k8s —> swarm
warm	▽ − ▲
cold	▽ − ▲

(b) Throughput

	k8s —> swarm
warm	− − −
cold	− − −

(c) Success Rate

Table 5.27: Wilcoxon (Warm & Cold) Sequential Auto Scaling Enabled

Table 5.27 represents the *Wilcoxon* results of sequential workloads with auto scaling disabled using the same replica settings for both warm and cold start test cases. The results were only statistically significant for throughput with p-values $<$ 5% on replica settings (1, 20) where K8S performed better on replica 20 and Docker Swarm on replica 1. However, both response time and success rate results were not statistically significant.

Chapter 6

Discussion

This chapter represents the discussion of the study based on our observed results after conducting the experiment for all test cases as illustrated in chapter 5 where we addressed all the possible combinations of the required test cases based on the scenarios and hypotheses defined in chapter 4. It also map the findings of the experiment to the hypotheses and research questions discussed before where each section is going to validate set of hypotheses and answer research questions. The OpenFaas framework selected in this study has the flexibility to integrate with different container orchestrators easily by implementing the openfaas provider which gives the ability to talk natively to each of the defined container orchestrators that allows and support all required operations for deploy, create, scale, manage and place any Serverless function inside the container orchestrator cluster.

6.1 Impact of Container Orchestrators & OpenFaas Providers on Function Performance

Looking back to the results we discussed on chapter 5, it was clear enough that all the test cases results were not the same across all defined container orchestrators (Docker Swarm, Kubernetes, Nomad) where performance of deployed Serverless functions were varied for each container orchestrator as the preferability went to Docker Swarm. Each container orchestrator has its own way of managing functions (containers) inside cluster from routing, networking connectivity and scheduling function as running container. Moreover, in order to invoke all the operations required for managing functions from OpenFaas perspective, different implementing of OpenFaas providers are supported to achieve this goal where each container orchestrator has different approach of managing functions varied from crud operations, routing and scaling. The next two subsections will focus on both OpenFaas providers and container orchestrators architecture.

6.1.1 OpenFaas Providers Architecture

The entry point to manage and communicate with Serverless function for any of the container orchestrator cluster is from the *OpenFaas Gateway* which is a required component that need to be deployed inside the cluster. However, each container orchestrator defined in this study has its own way of managing, scheduling functions that can simply run a container inside the container orchestrator cluster .

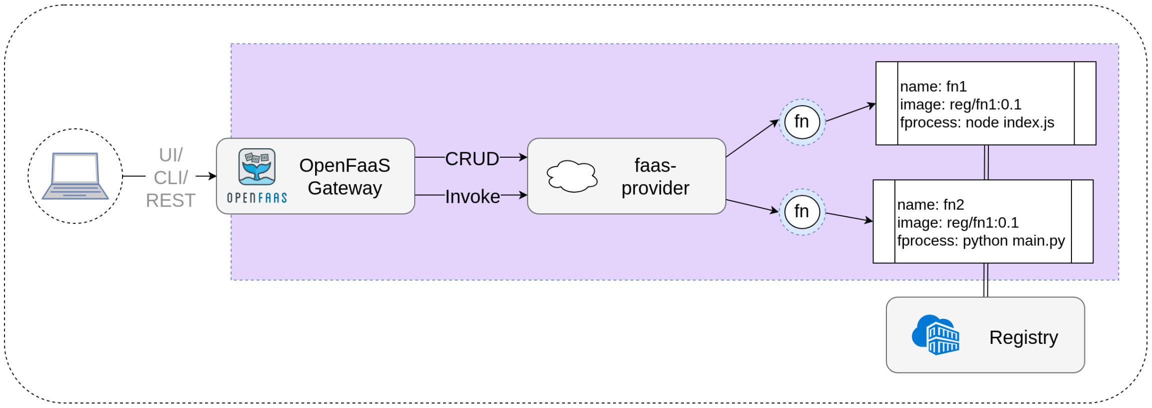


Figure 6.1: OpenFaas Provider Conceptual Diagram [60]

Figure 6.1 shows a conceptual diagram of how OpenFaas framework manage functions when deploy them to container orchestrator cluster where OpenFaas forward all the requests to the the *faas-provider* which is responsible for managing Serverless functions inside the cluster. *faas-provider* has different implementations based on container orchestrator.

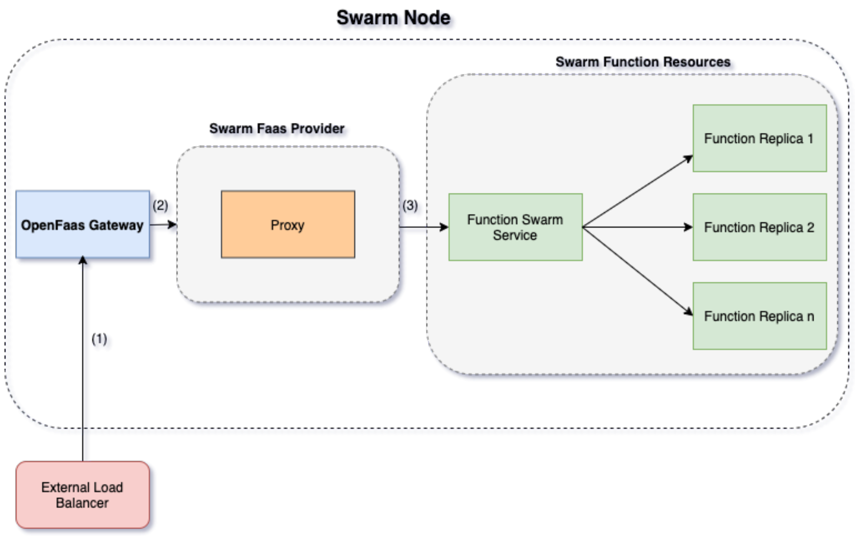


Figure 6.2: OpenFaas Swarm Provider Diagram

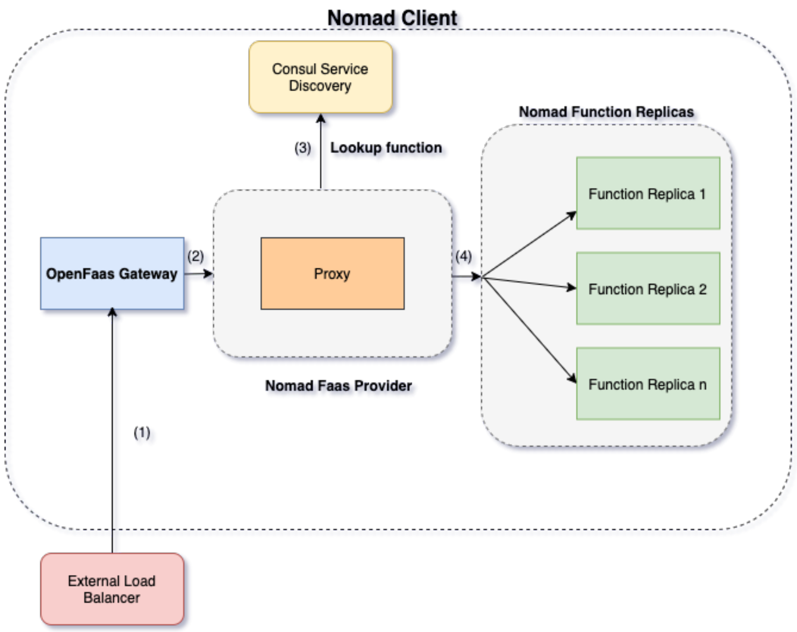


Figure 6.3: OpenFaas Nomad Provider Diagram

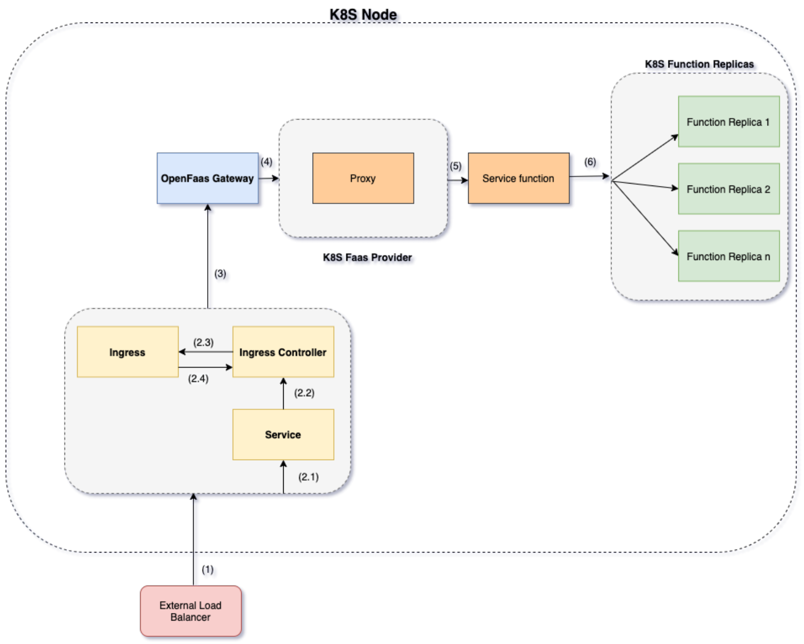


Figure 6.4: OpenFaas K8S Provider Diagram

Figures 6.2, 6.3 and 6.4 represent the flow of all OpenFaas providers that used to communicate with all defined container orchestrators where the communication for management and http function operations were done using RESTful calls where all these figures represent the routing flow of requests starting from outside clusters to reach the target deployed Severless function and its noticeable that the whole figures has *proxy* component which represents part of functionality of *faas-provoider* where it handle other operations related to function management inside the container orchestrator cluster. External load balancer (HAProxy) distributes the loads between all container orchestrator nodes where the OpenFaas Gateway is the entry point of accepting the incoming requests and after that it redirects them to the *faas-provider* to handle routing to the deployed function.

Figure 6.2 represents the *faas-provider* for Docker Swarm where the swarm provider accepts the requests from OpenFaas Gateway and then based on the function name it will do a DNS lookup for the function service that is going to distribute the loads between function replicas in case there are more than one replica exists. On the other hand, Figure 6.3 represents the *faas-provider* for Nomad that also accepts the requetss from OpenFaas Gatewat and then it lookups the function name using service discovery (Consul) where each deployed function will be registered with the service discovery so that it can be used later on when requested by the Nomad provider where it handles the round robin distribution between function replicas if exists. Finally, Figure 6.4 represents the *faas-provider* for K8S where routing requests is more complicated than Docker Swarm and Nomad. Ingress Controller is deployed inside the K8S cluster where it handles the communication between the external load balancer and the OpenFaas gateway where it accepts the requests from a deployed service inside the cluster and then the ingress controller will do the mapping by calling the ingress object to provide the ingress controller with the endpoint of the OpenFaas gateway. Once requests reach out the OpenFaas gateway it will redirect them to the K8S *faas-provider* to communicate with the function using a service object in order to match the function name and forward all requests to the desired function replicas.

6.1.2 Container Orchestrators Architecture

All the container orchestrators selected for this study mainly used for docker based applications. However, Docker Swarm is the simplest orchestrator among them where it only supports Docker. On the other hand, Kubernetes is more a complex system that ship with a collection of components tied together to provide its full functionality where it can also run Rkt [1] based applications. Moreover, Nomad is a general purpose solutions that supports containerized, virtualized and standalone applications using different drivers.

The architecture of Nomad & Docker Swarm are much simpler than Kubernetes in terms of component numbers and how they interact with each others. Each container orchestrator uses different scheduling method in order to assign and run a workload to one of the cluster members. Invoking scheduler of container orchestrator for OpenFaas framework is triggered indirectly when placing function inside any of the container orchestrator cluster by using OpenFaas Gateway API which get invoked using OpenFaas ClI or automatically if auto scaling setting of Serverless function is enabled. During our experiment, enable auto scaling is one of the settings used for all test cases so that functions can be created on fly based on workload requests. As we mentioned, that placing new function can be done via OpenFaas API gateway but the allocation and running the deployed function is the responsibility of container orchestrator where each one handle it differently.

[1]Rkt: `https://coreos.com/rkt/`

Nomad, starts the scheduling whenever jobs get created that represent the specifications of tasks (functions) where Nomad should run. Upon job is created, updated or de-registered, an evaluation is created based on job submission events and then enqueued into an evaluation broker to be processed later on by a scheduling workers where they are responsible for de-queuing evaluations from the broker and then to invoke the relevant scheduler based on job specification. Processing an evaluation is done by the scheduler where allocation plan is generated which contains set of allocations to create and update. The allocation consists of two main phases, feasibility checking and ranking where the first phase responsible for filtering unhealthy nodes and the second phase scores feasible nodes to find the best fit node based on bin packing. The highest rank node is selected by the scheduler to assign the required workloads to be executed. On the other hand, *Kubernetes* starts the scheduling for new incoming pod objects that wraps the desired functions (containers) or whenever unscheduled pods are available. The component responsible for scheduling is called *kube-scheduler* which run as part of the control plane where each pod has different requirements from other pods as *kube-scheduler* used the filtering and scoring approach that similar to Nomad but with less complexity. The *kube-scheduler* selects the best nodes based on scoring approach which meets the pod requirements and once that is done it will notify the API Server. Finally, *Docker Swarm* starts the scheduling when the service is submitted which represents the tasks (functions) definitions to be executed on the cluster nodes where Docker Swarm used a strategy called "spread" which tries to allocate a service task after do an assessment process of the available resources for cluster nodes. This simple strategy means that tasks allocations are evenly spread across the nodes inside the cluster.

Once the functions are allocated inside the cluster, it must be discoverable so that routing to these function can be achieved. On the previous section we mentioned how routing is implemented on each OpenFaaS provider where *Nomad* using service discovery approach by registering the function once get deployed inside the cluster by integrating Nomad with service networking solution consul that keep records for functions. On the other hand, both *Docker Swarm* & *Kubernetes* used DNS server for functions resolution.

Based on our discussion for OpenFaas providers and container orchestrator architecture and how they differently handle routing, scheduling for Serverless functions inside cluster and based on the results we obtained from chapter 5 emphasizes our H_1-H_3 hypotheses and our main research question **RQ1)** about the relationship of Serverless functions and container orchestrators and their impacts on the functions performance for response time, throughput and success rate where we able to validate them statistically using *Wilcoxon* tests that confirmed our first three hypotheses and allowed us to reject the Null hypotheses for all metrics

6.2 Workload Requests & Serverless Function Performance

Parallel and sequential are the two types of workloads that is used during this study which combined with different settings with all test cases. There was a noticeable differences between the results of the sequential workloads and parallel workloads for all test scenarios across all container orchestrators because of the requests volume generated per second was different. However, despite the differences, the performance results for both parallel and sequential workloads were not the same for all container orchestrators where parallel workloads were used with both enabled & disabled auto scaling whereas sequential workloads were used only with disabled auto scaling that set the number of replicas manually. Performance of Serverless functions under parallel workloads with auto scaling enabled showed better performance comparing parallel & sequential workloads with auto scaling disabled and that also was noticeable between all the defined container orchestrators as the Docker Swarm generally had the best result and then it comes K8S and Nomad. The results obtained from chapter 5 were aligned with our H_4-H_6 hypotheses and answer to our second research question **RQ2)** about how different workloads can be impact the performance of deployed Serverless functions from one container orchestrators to another where response time, throughput and success rate were affected when using different container orchestrators using the same settings for the same function and we were able to validate that statistically using *Wilcoxon* test.

6.3 Computational Requirement & Serverless Function Performance

Multiple Serverless functions were used in this study that represent different operation types related to CPU/Memory, Network, I/O and chaining functions.

6.3.1 I/O Operation

The I/O operation results obtained in chapter 5 showed a clear image about the difference between all container orchestrators where throughput of the same deployed Serverlesss function had different results as the Docker Swarm had the best results for parallel and sequential workloads with all combinations of test cases. On the other hand, the response time also had different results for the same deployed Serverless function where the parallel workloads test cases with auto scaling enabled showed a clear differences in response time results in all container orchestrators and how they reacted to high workloads where also Docker Swarm had the best result of handling the high loads. Moreover, the parallel workloads with auto scaling disabled show also outperforming of Docker Swarm with handling requests to all created Serverless function replicas. The results of sequential workloads between all container orchestrators were close to each other where K8S & Docker Swarm nearly had the same results and Nomad had the worst among them. Finally, the success rate results were identical for all test cases except Nomad had 7% error rate in parallel workloads with auto scaling enabled. The overall results showed that Docker Swarm performed better than others at least for two defined performance metrics response time and throughput and its worth to mention that K8S came in second place after Docker Swarm in terms of which container orchestrators had the best results and Nomad had the worst one.

6.3.2 CPU/Memory Operation

The CPU/Memory operation is defined in the two tests scenarios: Computation & Chaining Serverless Functions. The results from the computation scenario showed a clear difference between the container orchestrators where throughput and response time of the same deployed function had different results among all container orchestrators where Nomad had better performance for parallel workloads with auto scaling enabled than Docker Swarm & K8S. However, Docker Swarm exceeds Nomad and had better results on parallel workloads with auto scaling disabled where it seems that the number of replicas on Nomad was not enough to handle all the requests or there was unexpected issue for replica initialization encountered. The results of sequential workloads between all container orchestrators were close to each other but Docker Swarm had the best result obtained there. Finally, the success rate results were identical for all test cases between all container orchestrators with exceptions in Nomad that had some error rate 4% for parallel workloads with auto scaling enabled test cases. The overall results showed that Nomad had better results in parallel workloads with auto scaling enabled and Docker Swarm in remaining test cases at least for two defined performance metrics of response time and throughput.

The results from the Chaining Serverless Functions had two functions talk to each other via HTTP requests where one of the function (destination) is the same function (Matrix multiplication) used on CPU/Memory operation and the (source) function accepts the requests generated by the *faas-exp* and forward the payload to the destination function to return results back to the source function. Adding extra layer of communication to call CPU/Memory function affected the results where Docker Swarm this time had best results for throughput and response time specially in parallel workloads with auto scaling enabled and performed nearly better than others for sequential & parallel with auto scaling disabled. However, the success rate was identical for all container orchestrators in sequential workloads with 100% success rate. Moreover, only Docker Swarm and K8S had similar results on parallel workloads with auto scaling enabled/disabled as Nomad had high error rate of handling chaining functions with error rate varied from 4% - 45%. The overall results showed that Docker Swarm had better results in all test cases at least for two defined performance metrics of response time and throughput.

6.3.3 Network Operation

The network operation represents a Serverless function that connect to FTP server in order to download a file located there. The results obtained for network operation showed that not all container orchestrators had the same result and there was clear variation in results especially in success rate metric for high workloads with auto scaling enabled and that could be because of multiple reasons. First reason related to the capacity of the FTP server of how it can handle high workloads. The second one related to the ability of container orchestrator provider to serve all submitted requests. The third reason is about the deployed OpenFaas gateway was unable to serve all requests which were forwarded from the external load balancer. Comparing the results of other metrics like response time and throughput, its noticeable the differences on the parallel workloads for the all container orchestrators. The results obtained from sequential workloads showed that nearly all container orchestrators were close to each other with slight differences. The overall results showed that Nomad had better results in terms of response time and throughput at least for the parallel workloads but it showed also a bad results for success rate.

Based on the obtained results from the Computation & Chaining Serverless Functions scenarios, the performance of deployed Serverless function for any operation defined has different result for performance at and that was aligned with our H_7-H_9 hypotheses and answer to our main research question **RQ3)** that assumes a relationship between the computational requirements of Serverless function and using different container orchestrator for all performance metrics defined for this study where we were able to validate that statistically using *Wilcoxon* test .

6.4 Programming Languages/Runtimes & Serverless Function Performance

Multiple programming languages/runtimes were used in this study in order to find if using different programming languages/runtimes to deploy Serverless function to different container orchestrators can impact the performance of the function. Four programming languages/runtimes were used where two of them were complied languages (Java, Go) and the other two (Python, NodeJS) were interpreted.

6.4.1 Compiled Programming Languages

The obtained results for using Java programming language showed that with high parallel workloads (5 - 20) & sequential workloads the response time and throughput for two container orchestrators (K8S, Docker Swarm) were nearly the same except with Nomad which had better results for both metrics. However, with higher parallel workloads (50 users) the Nomad had worst results than others where Docker Swarm had the best result. The success rate for all test cases had identical results except with one container orchestrator (Nomad) that had the worst result among them. On the other hand, the obtained results for using Go programming language showed that response time and throughput were not the same for the parallel workloads in all cases where the best achieved result was on Docker Swarm and then for K8S. However, the success rate for Nomad had the worst result among all test cases for both parallel and sequential workloads where error rate reached a high value 68% despite the fact that for this function on Nomad the number of iteration runs reached out to 12 and still had the same bad result where the root cause for this high error rate because of bad gateway responses that prevent from forwarding the requests to target function. Finally, the results of sequential workloads for both Docker Swarm and K8S nearly the same and close to each other with preferability to Docker Swarm.

The overall results showed that there was a variation in obtained results in terms of all performance metrics where one container orchestrator (Docker Swarm) had better results than others. On other hand, the code execution of Go & Java functions are different where Go is somehow similar to C functions that do the compilation to machine code and run it directly. However, Java has an additional step where the generated byte code by the JVM has to be compiled to machine code using just-in-time (JIT) compiler before get running. With this given fact, the results for Java in all container orchestrators were better than Go and this is because of the simplicity of the function written for the experiment between programming languages. Moreover, the high error rate related to Go functions in Nomad is mainly related to Bad gateway error which need further investigation.

6.4.2 Interpreted Programming Languages

The obtained results for using Python programming language & NodeJS(Javascript runtime) showed that with parallel workloads for all test cases the results between all container orchestrators were not the same for response time and throughput where the best results achieved by the Docker Swarm, K8S and then Nomad. On the other hand, the success rate also were only identical for all container orchestrators except Nomad that had some error rate reached to 29% for NodeJS function. Finally, the results of sequential workloads for both Docker Swarm and K8S nearly the same and close to each other with preferability to Docker Swarm. The overall results showed that there were variations between container orchestrators where one container orchestrator (Docker Swarm) achieved better results than others. Moreover, the selected interpreted languages/runtimes in this case have different behaviours and architectures that could affect on the results where NodeJS is built based on Chrome's V8 engine and supports handling multiple requests at the same time because of the event-driven non-blocking architecture. However, Python is a single-flow where requests get processed much slower. Moreover, Python does not support real multi-threading because of the usage Global Interpreter Lock (GIL)

Based on the obtained results from both compiled and interpreted programming languages, its clear that the performance of deployed Serverless function using different programming languages on different container orchestrators had different results and noticeable impact on performance which is something aligned with our H_{10}-H_{12} hypotheses and provides answer to our main research question **RQ4)** about the impact of programming languages/runtimes of deployed Serverless function across different container orchestrators where the statistical tests we applied for this purpose using *Wilcoxon* confirmed these hypotheses.

6.5 Warm/Cold Start & Serverless Function Performance

This section represents the results obtained from running function with warm start following by cold start where functions will be idled for certain amount of time as specified in chapter 5. Running the function with cold start means that there will be a some extra latency added before serving the request for unavailable function as the number of function replicas will be scaled to 0. OpenFaas has the ability to do a "zero-scale" with the help of component called *faas-idler* which can be deployed to the container orchestrators that is responsible of reducing the function to zero replica. Moreover, the function replicas can be brought back again to the required amount of replicas when its needed by the *faas-provider* component. Figure 6.5 represents the flow of how *faas-idler* works and Listing 4 represents a psuedo code for faas-idler.

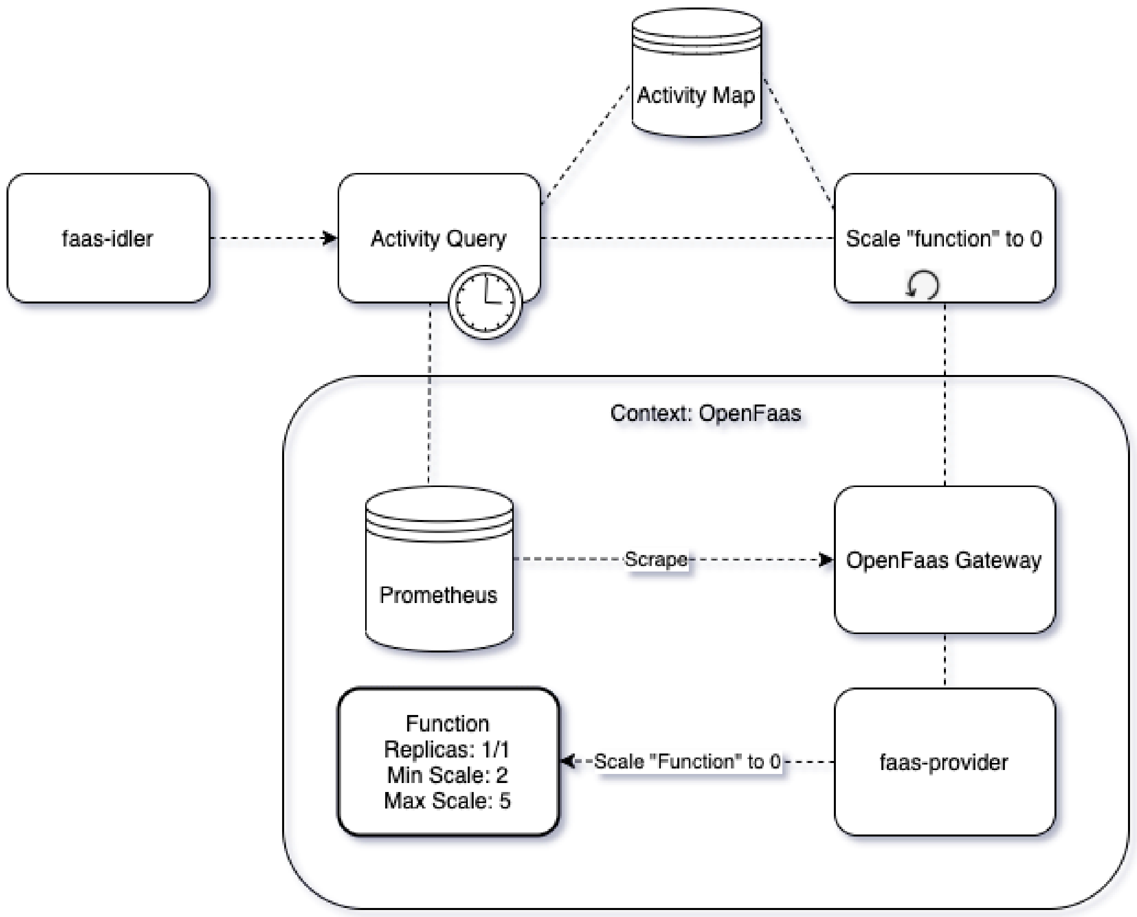


Figure 6.5: Conceptual diagram of faas-idler [68]

```
while True:
  idle = query_prometheus_for(functions idle over N minutes)
  for function in idle:
      gateway.scale(0, function)

  sleep_for(interval)
```

Listing 4: Psuedo Code of faas-idler.

The *faas-idler* is keep pulling all metrics from the prometheus [2] component to make sure if the function has been idle for a given amount of time so that it can be scaled down . On the other hand, if there is an incoming request for the idled function that means to scale up function to the required amount of replicas so that it can serve the request. Figure 6.6 represents the flow for serving requests of a Serverless function on idled state which requires to scale up the function from zero to the required number of replicas. This introduces a latency as the upstream proxy (faas-provider) will need to check if there is a function ready for serving requests before forwarding the them to the target function as it will invoke API gateway for scaling function to the required number of replicas.

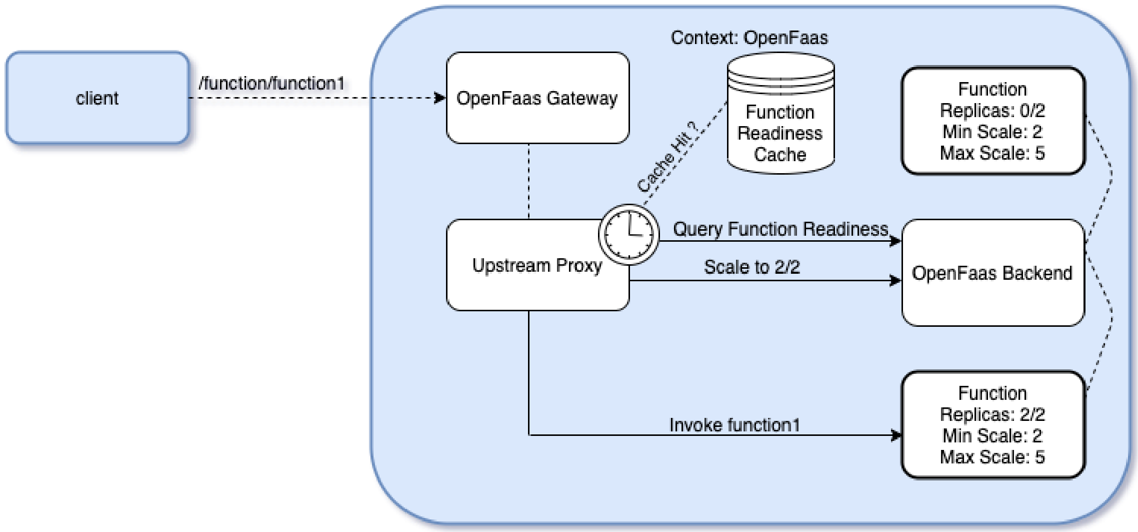


Figure 6.6: Conceptual Diagram of OpenFaaS scaling Up [68]

The results obtained for warm/cold scenario from as specified in chapter 5 only relevant for Docker Swarm and K8S as Nomad was disqualified for technical issues on deploying *faas-idler*. The results for warm start requests showed that Docker Swarm had the best results for both parallel and sequential workloads for all test cases at least for two performance metrics: response time and throughput. On the other hand, K8S had better results for parallel workloads than Docker Swarm with concurrency levels (15, 20, 50) in terms of response time where Docker Swarm achieved better results in other concurrency levels (5, 10).

[2]prometheus: `https://prometheus.io/`

Based on the obtained results from warm & cold start scenario showed that the results were not the same between Docker Swarm and K8S where at least two performance metrics were different as Docker Swarm achieved better results than K8S in most test cases. The observed results showed that response time and throughput were affected by using different container orchestrators as results were different for both cold and warm start. The throughput results of cold start comparing to the warm start were much smaller because of the extra latency added and since enforcing the cold start will not be maintained long time because the number of ready replicas will be available after scaling up and that explains why the cold response time were close to the warm start and sometime smaller. The results we obtained from applying median and *Wilcoxon* tests for both cold & warm start showed that the impact was limited to response time and throughput only which aligned with H_{13} & H_{14} and answer to our main research question **RQ5)**. However, the success rate for all test cases in this scenario were not statistically significant.

Chapter 7

Conclusion

7.1 Conclusions

The era of Severless computing became an attractive topic recently in IT industry and academia specifically for IoT and Edge Computing researches. Studying performance of Serverless is a common topic in literature review where it mainly focused on studying the Serverless on public cloud using managed Serverless services like AWS Lambda, Google Cloud Functions, Azure Functions and more like the works presented by [113] [95] [103] [100]. Most of the works focused on performance metrics like response time & throughput where few of them illustrated how they conducted their experiments by using some custom benchmark tools they developed for that purpose. On the other hand, there were few interesting researches that studied Serverless on on-premise infrastructure using open source Serverless frameworks like OpenFaaS, Kubeless and Fission where very limited researches studied the performance of Serverless functions that using single container orchestrators like K8S as presented by [86] [105]. This study focused on studying the performance of Serverless using different container orchestrators like Docker Swarm, K8S and Nomad as no previous works tried to studied the impact of the different container orchestrators on the performance of deployed Serverless functions. Moreover, none of the previous works that studied the Serverless on on-premise infrastructure used multiple programming languages/ runtimes, complex operations like: CPU/Memory, I/O or studied the impact of cold start. Finally, none of the previous works either public or on-premise studies illustrated fully details of how to conduct, aggregate, analyze and visualize results for their studies.

Based on our knowledge, this is the first research that studied the impact of container orchestrators on Serverless functions performance with large number of test cases that cover different scenarios using various types of computations and different programming languages/runtimes.

Based on the obtained results from this study, the following conclusion can be drawn.

- **Container orchestrators have impacted the performance of Serverless function**: The results we obtained from conducting this study showed that using different container orchestrators have impacted the performance of Serverless function where it consider to be a good insight for anyone want to use Serverless functions with container orchestrators for different purposes especially for IoT and edge computing applications where generally Docker Swarm performed better than other container orchestrators.
- **The performance of Servelress function that used different generated workloads have impacted by container orchestrators**: In this study we generated two types of workloads (parallel, sequential) with the capability of auto scaling provided by the OpenFaas framework. The results obtained from the generated test cases were not the same for the parallel & sequential workloads where it reached 9X-10X performance in sequential workloads for Serverless function written in Java and deployed to Nomad. Moreover, using different concurrency levels for parallel workloads have noticeable impact on performance metrics.
- **The performance of Servelress function based on different computation have impacted by container orchestrators**: Our results showed that implementing Serverless function with different operation types had different results between container orchestrators where deploying Serverless function with I/O operation were varied and the Docker Swarm performed better than others. On the other hand, deploying Serverless function with CPU/Memory operation had been racing between Nomad & Docker Swarm especially in parallel workloads. finally, deploying Serverless function with network operation had nearly the same results across all test cases except on Nomad where it achieved the worst success rate between them.

- **The performance of Servelress function written in different programming languages/runtimes have impacted by container orchestrators**: Our results showed that implementing Serverless function using interpreted languages like Python & NodeJS have different results between container orchestrators where Docker Swarm performed better than others. On the other hand, the complied language like Java, Nomad performed better than others. However, the other complied language (GO) had better performance on Docker Swarm instead. These results could help in selection which programming language/runtime to use with which container orchestrator.
- **The performance of Servelress function in warm & cold started cases have impacted by container orchestrators**: The results of this study showed that both response time & throughput have impacted by container orchestrators for warm & cold start test cases especially in parallel workloads where the preferability goes to Docker Swarm.

7.2 Threats to Validity

On this study our main goal is to find if there is a relationship between the performance of the Serverless function and container orchestrators. We were able to validate that based on the set of independent variables defined for this study where our candidate Serverless framework selected for this purpose was OpenFaas that has different interfaces for each container orchestrator responsible for routing, placing container and function management inside cluster. Moreover, the internal architecture of the framework itself where it contains several component like OpenFaas Gateway which acts as an entry point of the framework in order to handle routing for provider based on the selected container orchestrator. These factors could have impacts on dependent variables which measure the performance of Serverless function. On the other hand, we evaluated 9 Serverless frameworks so that we can judge better which is the suitable Serverless framework that we can use in this study based on different criteria like supporting multiple container orchestrators where we did an experiment to run each selected Serverless frameworks that support multiple container orchestrators using simple hello world function but we did not conduct the same test cases that we did for OpenFaas to other Serverless candidates frameworks so that we can compare their results with the one we got from running OpenFaas to make sure if we can generalize our results to other Serverless frameworks. This study focused on studying the performance of Serverless functions using specific metrics that includes response time, throughput and success rate because they considered to be critical factors in real software environment which could have hundreds of Serverless functions under different configuration and settings with variation on function implementation logic, programming languages, frequency of functions invocation which explains the reasons for including a lot factors where we able to measure and validate what we proposed for the relationship between all factors and the performance metrics. However, since we are using different combinations of settings between factors during our experiment, this could be a threat to our construct validity that may add some ambiguity about determining which factor causes the impact. Finally, we were able to measure and validate the relationship between defined variables statistically using *Wilcoxon* tests where our assumption aligned with statistical tests we applied during the experiment and we can conclude that the value of static power is relatively high since we reject the Null hypothesis in most of the test cases which minimize from threats to our conclusion validity.

7.3 Future Works

We believe there are further opportunities to extend and continue on this research on the following areas:

- **Conducting more complex test cases**: We can extend the complexity of the test cases by simulating a real software application that can deal with database transactions, external communication with other systems and increase the number of loads to thousands of users.
- **Add support for asynchronous functions**: All the conducted test cases were based on synchronous operations where deploying function that use message queues is doable as most of the Serverless frameworks support that.
- **Use test cases that cover authentication and authorization scenarios**: We ignore the authentication and authorization for all our test cases which can be added in future to measure the impact of enabling them on performance.
- **Deploy machine learning models**: Deploying Serverless function that supports machine learning models is also another work that can be added to see how machine learning model can behave among different container orchestrators when get deployed as part of Serverless functions.

References

[1] [Online]. Available: https://openwhisk.apache.org/documentation.html.

[2] [Online]. Available: https://docs.fission.io/docs/.

[3] [Online]. Available: https://openwhisk.apache.org/documentation.html#actions-creating-and-invoking.

[4] [Online]. Available: https://openwhisk.apache.org/documentation.html#programming-model-triggers.

[5] *Apache/openwhisk*. [Online]. Available: https://github.com/apache/openwhisk.

[6] *Apache/openwhisk*. [Online]. Available: https://github.com/apache/openwhisk/blob/master/LICENSE.txt.

[7] *Apache/openwhisk*. [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/metrics.md.

[8] *Create new functions*. [Online]. Available: https://docs.openfaas.com/cli/templates/#template-store.

[9] *Docker swarm*. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/.

[10] *Fission auto-scaling*. [Online]. Available: https://fission.io/features/.

[11] *Fission cli*. [Online]. Available: https://docs.fission.io/docs/installation/.

[12] *Fission/fission*. [Online]. Available: https://github.com/fission/fission/blob/master/LICENSE.

[13] *Fission/fission*. [Online]. Available: https://github.com/fission/fission.

[14] *Fn project*. [Online]. Available: https://github.com/fnproject/docs/blob/master/fn/general/introduction.md.

[15] *Fnproject/docs*. [Online]. Available: https://github.com/fnproject/docs/blob/master/fn/operate/runner_pools.md.

[16] *Fnproject/docs.* [Online]. Available: https://github.com/fnproject/docs/blob/master/fn/develop/triggers.md.

[17] *Fnproject/fn.* [Online]. Available: https://github.com/fnproject/fn/blob/master/LICENSE.

[18] *Fnproject/fn.* [Online]. Available: https://github.com/fnproject/fn/tree/master/examples/grafana.

[19] *Hashicorp/faas-nomad.* [Online]. Available: https://github.com/hashicorp/faas-nomad.

[20] *Iron-io/functions.* [Online]. Available: https://github.com/iron-io/functions.

[21] *Iron-io/functions.* [Online]. Available: https://github.com/iron-io/functions/blob/master/LICENSE.

[22] *Iron-io/functions.* [Online]. Available: https://github.com/iron-io/functions/blob/master/docs/faq.md#which-languages-are-supported.

[23] *Iron-io/functions.* [Online]. Available: https://github.com/iron-io/functions/tree/master/examples/hello.

[24] *Iron-io/functions.* [Online]. Available: https://github.com/iron-io/functions/tree/master/docs/operating.

[25] *Iron-io/functions.* [Online]. Available: https://github.com/iron-io/functions/blob/master/docs/operating/metrics.md.

[26] *Knative auto-scaling.* [Online]. Available: https://knative.dev/docs/serving/configuring-the-autoscaler/.

[27] *Knative cli.* [Online]. Available: https://github.com/knative/client.

[28] *Knative monitoring.* [Online]. Available: https://knative.dev/docs/serving/installing-logging-metrics-traces/.

[29] *Knative on kubernetes.* [Online]. Available: https://knative.dev/docs/.

[30] *Knative serving code samples.* [Online]. Available: https://knative.dev/docs/serving/samples/index.html.

[31] *Knative triggers.* [Online]. Available: https://knative.dev/docs/eventing/sources/.

[32] *Knative/serving.* [Online]. Available: https://github.com/knative/serving/blob/master/LICENSE.

[33] *Knative/serving.* [Online]. Available: https://github.com/knative/serving/.

[34] *Kubeless auto-scaling.* [Online]. Available: https://kubeless.io/docs/autoscaling/.

[35] *Kubeless cli.* [Online]. Available: https://kubeless.io/docs/quick-start/.

[36] *Kubeless monitoring.* [Online]. Available: https://kubeless.io/docs/monitoring/.

[37] *Kubeless runtime variants.* [Online]. Available: https://kubeless.io/docs/runtimes/.

[38] *Kubeless triggers.* [Online]. Available: https://kubeless.io/docs/implementing-new-trigger/.

[39] *Kubeless/kubeless.* [Online]. Available: https://github.com/kubeless/kubeless/blob/master/LICENSE.

[40] *Kubeless/kubeless.* [Online]. Available: https://github.com/kubeless/kubeless.

[41] *Kubernetes components.* [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/.

[42] *Kubernetes-native serverless.* [Online]. Available: https://kubeless.io/docs/.

[43] *Kyma - an easy way to extend enterprise applications on kubernetes.* [Online]. Available: https://kyma-project.io/docs/components/serverless#architecture-architecture.

[44] *Kyma auto-scaling.* [Online]. Available: https://kyma-project.io/docs/0.9#installation-install-kyma-locally-enable-horizontal-pod-autoscaler-hpa.

[45] *Kyma monitoring.* [Online]. Available: https://kyma-project.io/docs/0.9#details-components-monitoring.

[46] *Kyma on kubernetes.* [Online]. Available: https://kyma-project.io/docs/root/kyma/#overview-in-a-nutshell.

[47] *Kyma triggers.* [Online]. Available: https://kyma-project.io/docs/components/application-connector#tutorials-trigger-a-lambda-with-events.

[48] *Kyma-project/kyma.* [Online]. Available: https://github.com/kyma-project/kyma/blob/master/LICENSE.

[49] *Nomad architecture*. [Online]. Available: https://www.nomadproject.io/docs/internals/architecture.html.

[50] *Nuclio auto-scaling*. [Online]. Available: https://nuclio.io/.

[51] *Nuclio cli*. [Online]. Available: https://nuclio.io/docs/latest/reference/nuctl/.

[52] *Nuclio documentation*. [Online]. Available: https://nuclio.io/docs/latest/reference/runtimes/.

[53] *Nuclio monitoring*. [Online]. Available: https://nuclio.io/docs/latest/tasks/configuring-a-platform/#metric-sinks-metrics.

[54] *Nuclio on kubernetes*. [Online]. Available: https://nuclio.io/docs/latest/setup/k8s/getting-started-k8s/.

[55] *Nuclio triggers*. [Online]. Available: https://nuclio.io/docs/latest/reference/triggers/.

[56] *Nuclio/nuclio*. [Online]. Available: https://github.com/nuclio/nuclio/blob/master/LICENSE.

[57] *Openfaas auto-scaling*. [Online]. Available: https://docs.openfaas.com/architecture/autoscaling/.

[58] *Openfaas deployment*. [Online]. Available: https://docs.openfaas.com/deployment/.

[59] *Openfaas gateway*. [Online]. Available: https://docs.openfaas.com/architecture/gateway/.

[60] *Openfaas provider conceptual model*. [Online]. Available: https://github.com/openfaas/faas-provider.

[61] *Openfaas triggers*. [Online]. Available: https://docs.openfaas.com/reference/triggers/.

[62] *Openfaas/faas*. [Online]. Available: https://github.com/openfaas/faas.

[63] *Openfaas/faas*. [Online]. Available: https://github.com/openfaas/faas/blob/master/LICENSE.

[64] *Openwhisk auto-scaling*. [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/README.md.

[65] *Openwhisk deployment*. [Online]. Available: https://openwhisk.apache.org/documentation.html#development_tools.

[66] *Projectodd/openwhisk-openshift*. [Online]. Available: `https://github.com/projectodd/openwhisk-openshift`.

[67] *Running ironfunctions in production*. [Online]. Available: `https://github.com/iron-io/functions/blob/c070e629e9e3210316b44e3372e7edd869c01425/docs/operating/production.md`.

[68] *Scale to zero and back again with openfaas*. [Online]. Available: `https://www.openfaas.com/blog/zero-scale/`.

[69] *Semode - serverless monitoring and debugging*. [Online]. Available: `https://github.com/johannes-manner/SeMoDe` (visited on 10/11/2019).

[70] *Stack*. [Online]. Available: `https://docs.openfaas.com/architecture/stack/`.

[71] R. Prasad and V. Rohokale, "Cyber security: The lifeline of information and communication technology," in. Springer Series in Wireless Technology, Jan. 2020, pp. 111–124, ISBN: 978-3-030-31702-7. DOI: `10.1007/978-3-030-31703-4_8`.

[72] *Apache/openwhisk-cli*, Nov. 2019. [Online]. Available: `https://github.com/apache/openwhisk-cli`.

[73] J. Butler, *Serverless patterns*, Apr. 2019. [Online]. Available: `https://medium.com/dropbox-design/are-app-reviews-worth-reading-518d3211872f`.

[74] *Docker overview*, Dec. 2019. [Online]. Available: `https://docs.docker.com/engine/docker-overview/`.

[75] *Fission monitoring*, Oct. 2019. [Online]. Available: `https://docs.fission.io/docs/installation/on-premise-install/`.

[76] *Fission triggers*, Oct. 2019. [Online]. Available: `https://docs.fission.io/docs/usage/trigger/`.

[77] *Fnproject/fn*, Nov. 2019. [Online]. Available: `https://github.com/fnproject/fn`.

[78] *Fnproject/fn*, Jan. 2019. [Online]. Available: `https://github.com/fnproject/fn/tree/master/examples/tutorial/hello`.

[79] *Fnproject/fn-helm*, Mar. 2019. [Online]. Available: `https://github.com/fnproject/fn-helm/`.

[80] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, and A. Palopoli, "Container orchestration engines: A thorough functional and performance comparison," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, May 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8762053.

[81] *Kyma-project/kyma*, Nov. 2019. [Online]. Available: https://github.com/kyma-project/cli.

[82] *Kyma-project/kyma*, Nov. 2019. [Online]. Available: https://github.com/kyma-project/kyma.

[83] J. Manner, G. Wirtz, M. Endreß, and T. Heckel, "Impact of application load in function as a service," Jun. 2019. DOI: 10.1109/UCC-Companion.2018.00054.

[84] *Nuclio/nuclio*, Nov. 2019. [Online]. Available: https://github.com/nuclio/nuclio.

[85] *Openfaas/faas-cli*, Nov. 2019. [Online]. Available: https://github.com/openfaas/faas-cli.

[86] A. Palade, A. Kazmi, and S. Clarke, "An evaluation of open source serverless computing frameworks support at the edge," May 2019. DOI: 10.1109/SERVICES.2019.00057.

[87] R. Pellegrini, I. Ivkic, and M. Tauber, "Function-as-a-service benchmarking framework," in *Proceedings of the 9th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, INSTICC, SciTePress, 2019, pp. 479–487, ISBN: 978-989-758-365-0. DOI: 10.5220/0007757304790487.

[88] (2019). What is edge computing ? [Online]. Available: https://www.hpe.com/emea_europe/en/what-is/edge-computing.html.

[89] T. Back and V. Andrikopoulos, "Using a microbenchmark to compare function as a service solutions," English, in *Service-Oriented and Cloud Computing*, K. Kritikos, P. Plebani, and F. De Paoli, Eds., ser. Lecture Notes in Computer Science, Springer, 2018, pp. 146–160, ISBN: 978-3-319-99818-3. DOI: 10.1007/978-3-319-99819-0_11.

[90] (Aug. 2018). Cncf serverless whitepaper v1.0, [Online]. Available: https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf.

[91] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pp. 000 149–000 154, 2018.

[92] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, and A. Iosup, "Serverless is more: From paas to present cloud computing," *IEEE Internet Computing*, vol. 22, no. 5, pp. 8–17, Sep. 2018. DOI: 10.1109/MIC.2018.053681358.

[93] K. Figiela, A. Gajek, A. Zima, B. Obrok, and M. Malawski, "Performance evaluation of heterogeneous cloud functions," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 23, e4792, 2018, e4792 cpe.4792. DOI: 10.1002/cpe.4792. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4792. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4792.

[94] *Iron-io/functions*, Aug. 2018. [Online]. Available: https://github.com/iron-io/functions.

[95] D. Jackson and G. Clynch, "An investigation of the impact of language runtime on the performance and cost of serverless functions," Dec. 2018, pp. 154–160. DOI: 10.1109/UCC-Companion.2018.00050.

[96] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 03, pp. 24–35, May 2018, ISSN: 1937-4194. DOI: 10.1109/MS.2018.2141039.

[97] A. R. S. Junior. (Nov. 2018). Runtime adaptation of microservices, [Online]. Available: https://repositorio.ufpe.br/handle/123456789/32395.

[98] J. Kim, T. J. Jun, D. Kang, D. Kim, and D. Kim, "GPU Enabled Serverless Computing Framework," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, Mar. 2018, pp. 533–540. DOI: 10.1109/PDP2018.2018.00090.

[99] K. Kritikos and P. Skrzypek, "A review of serverless frameworks," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, Dec. 2018, pp. 161–168. DOI: 10.1109/UCC-Companion.2018.00051.

[100] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, Jul. 2018, pp. 442–450. DOI: 10.1109/CLOUD.2018.00062.

[101] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, Apr. 2018, pp. 159–169. DOI: 10.1109/IC2E.2018.00039.

[102] M. Malawski, F. Kamil, G. Adam, and Z. Adam, "Benchmarking heterogeneous cloud functions," in. Feb. 2018, pp. 415–426, ISBN: 978-3-319-75177-1. DOI: 10.1007/978-3-319-75178-8_34.

[103] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, Dec. 2018, pp. 181–188. DOI: 10.1109/UCC-Companion.2018.00054.

[104] S. K. Mohanty, "Evaluation of serverless computing frameworks based on kubernetes," Jul. 2018. [Online]. Available: https://aaltodoc.aalto.fi/handle/123456789/33680.

[105] S. K. Mohanty, G. Premsankar, and M. D. Francesco, "An evaluation of open source serverless computing frameworks," Oct. 2018. DOI: 10.1109/CloudCom2018.2018.00033.

[106] A. Perez, G. Moltó, M. Caballer, and A. Calatrava, "Serverless computing for container-based architectures," *Future Generation Computer Systems*, vol. 83, Jan. 2018. DOI: 10.1016/j.future.2018.01.022.

[107] M. Sadaqat, R. Colomo-Palacios, and L. E. Knudsen, "Serverless computing: A multivocal literature review," 2018.

[108] S. Shillaker and P. R. Pietzuch, "A provider-friendly serverless framework for latency-critical applications," 2018.

[109] C. Völker. (Jun. 2018). Suitability of serverless computing approaches, [Online]. Available: https://elib.uni-stuttgart.de/handle/11682/9728.

[110] G. Adzic and R. Chatley, "Serverless computing: Economic and architectural impact," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, Paderborn, Germany: ACM, 2017, pp. 884–889, ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3117767. [Online]. Available: http://doi.acm.org/10.1145/3106237.3117767.

[111] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," in. Dec. 2017, pp. 1–20, ISBN: 978-981-10-5025-1. DOI: 10.1007/978-981-10-5026-8_1.

[112] M. Kalske. (Nov. 2017). Transforming monolithic architecture towards microservice architecture, [Online]. Available: http://hdl.handle.net/10138/234239.

[113] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," Jun. 2017, pp. 405–410. DOI: 10.1109/ICDCSW.2017.36.

[114] B. B. Rad, H. J. Bhatti, and M. Ahmadi, "An introduction to docker and analysis of its performance," *IJCSNS International Journal of Computer Science and Network Security*, vol. 173, p. 8, Mar. 2017.

[115] D. Sikeridis, I. Papapanagiotou, B. P. Rimal, and M. Devetsikiotis, "A comparative taxonomy and survey of public cloud infrastructure vendors," Oct. 2017.

[116] I. Baldini, P. Castro, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, and P. Suter, "Cloud-native, event-based programming for mobile applications," in *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, May 2016, pp. 287–288. DOI: 10.1109/MobileSoft.2016.063.

[117] P. Fawaz, S. Challita, Y. al-dhuraibi, and P. Merle, "Model-driven management of docker containers," Jul. 2016. DOI: 10.1109/CLOUD.2016.0100.

[118] M. H. Ferdaus, M. Murshed, R. N. Calheiros, and R. Buyya, "Network-aware virtual machine placement and migration in cloud data centers," in. May 2015, pp. 42–91, ISBN: 9781466682139. DOI: 10.4018/978-1-4666-8213-9.ch002.

[119] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15, Bordeaux, France: ACM, 2015, 18:1–18:17, ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741964. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741964.

[120] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb. 2013, pp. 233–240. DOI: 10.1109/PDP.2013.41.

[121] A. Prasanth, "Article: Cloud computing services: A survey," *International Journal of Computer Applications*, vol. 46, no. 3, pp. 25–29, May 2012.

[122] R. J. Creasy, "The origin of the vm/370 time-sharing system," *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 483–490, Sep. 1981, ISSN: 0018-8646. DOI: 10.1147/rd.255.0483.

[123] S. Mohanty, “Evaluation of serverless computing frameworks based on kubernetes,” en, G2 Pro gradu, diplomityö, 2018-08-20, p. 76. [Online]. Available: `http://urn.fi/URN:NBN:fi:aalto-201809034805`.

Appendix A

Serverless Functions

```
'use strict'

const fs = require('fs');

module.exports = async(event, context) => {
  console.log(event);
  console.log(context);
  let writer = await fs.createWriteStream('/tmp/log.txt', {
    flags: 'a' // 'a' means appending (old data will be preserved)
  });

  writer.write(event['body']);
  return context
    .status(200)
    .succeed(event['body'])
}
```

Listing 5: I/O Function

```
'use strict';

const clock = require('./clock.js');

exports.add = (m1, m2) => {
  const t0 = clock.clock();

  let result = [];
  for (let i = 0; i < m1.length; i++) {
    result[i] = [];
    for (let j = 0; j < m2[0].length; j++) {
      result[i][j] = m1[i][j] + m2[i][j];
    }
  }

  console.log("Matrix.add time: " + clock.clock(t0));

  //console.table(mResult)

  return result;
};

exports.create = (m, n) => {
  const t0 = clock.clock();

  let matrix = [];
  for (let i = 0; i < m; i++) {
    let row = [];
    for (let j = 0; j < n; j++) {
      // initialized 'random'
      row[j] = i * j;
    }
    matrix[i] = row;
  }

  console.log("Matrix.create time: " + clock.clock(t0));

  //console.table(matrix)

  return matrix;
};

exports.multiply = (m1, m2) => {
  const t0 = clock.clock();
  let result = [];
  for (let i = 0; i < m1.length; i++) {
    result[i] = [];
    for (let j = 0; j < m2[0].length; j++) {
      let sum = 0;
      for (let k = 0; k < m1[0].length; k++) {
        sum += m1[i][k] * m2[k][j];
      }
      result[i][j] = sum;
    }
  }

  console.log("Matrix.multiply time: " + clock.clock(t0));

  //console.table(mResult)

  return result;
};

exports.subset = (m1, offset_x, offset_y, width, height) => {
  const t0 = clock.clock();

  let result = [];
  for (let i = offset_x; i < m1.length && i < offset_x + width; i++) {
    result[i] = [];
    for (let j = offset_y; j < m1[0].length && j < offset_y + height; j++) {
      result[i - offset_x][j - offset_y] = m1[i][j];
    }
  }

  console.log("Matrix.subset time: " + clock.clock(t0));

  //console.table(mResult)

  return result;
};
```

Listing 6: Matrix Helper

```
'use strict'

const Matrix = require("./matrix.js");

module.exports = async(event, context) => {
  console.log(event);
  console.log(context);
  let param = 1;
  const isExist = 'param' in event['query'];
  if (isExist) {
      param = event['query']['param']

  }
  console.log('Input param=' + param);
  const a = await Matrix.create(param, param);
  const b = await Matrix.create(param, param);
  const resultBig = await Matrix.multiply(a, b);
  const result = {
    'result': await Matrix.subset(resultBig, 0, 0, 10, 10)
  };

  return context
    .status(200)
    .succeed(result)
}
```

Listing 7: Matrix Function

```
'use strict'
const ftp = require("basic-ftp");
const fs = require('fs');

module.exports = async (event, context) => {

  const ftpHost = process.env.ftp_host;
  const ftpUser = process.env.ftp_user;
  const ftpPassword = process.env.ftp_password;

  if (!ftpHost) {
      return context.status.fail('ftp Host is missing')
  }
  if (!ftpUser) {
      return context.status.fail('ftp User is missing')
  }
  if (!ftpPassword) {
      return context.status.fail('ftp Password is missing')
  }

  let result = {};
  let ftpResult = await ftpHandler(ftpHost, ftpUser, ftpPassword);
  let statusCode = 200;
  if (!ftpResult) {
      statusCode = 400;
      ftpResult = 'Error on downloading file from FTP'
      result['error'] = ftpResult
  } else{
      result['status'] = 'done';
  }

  return context
    .status(statusCode)
    .succeed({'result': result})

}

async function ftpHandler(host, user, password) {
    let result = null;
    const client = new ftp.Client();
    client.ftp.verbose = true;
    try {
        await client.access({
            host: host,
            user: user,
            password: password,
        });
         let writer = fs.createWriteStream('/tmp/test.txt', {
            flags: 'w' // 'a' means appending (old data will be preserved)
          });
        await client.downloadTo(writer, "test.txt");
        result = true
    }
    catch(err) {
        console.log(err)
        result = false
    }
    client.close();
    return result;
}
```

Listing 8: Network Function

```
'use strict';
const request = require('request');


function doRequest(headers, url, data) {
  return new Promise(function (resolve, reject) {
    request.post({
      headers: headers,
      url:     url,
      body:    data
    }, function (error, res, body) {
      if (!error && res.statusCode == 200) {
          console.log(body);
          console.log(res);
        resolve(body);
      } else {
         console.log(error);
        reject(error);
      }
    });
  });
}

module.exports = async(event, context, callback) => {
  const gateway_endpoint = process.env.gateway_endpoint;

  if (!gateway_endpoint) {
      return context.status.fail('Gateway URL is missing')
  }
  let param = 1;
  const isExist = 'param' in event['query'];
  if (isExist) {
      param = event['query']['param']

  }
  // This call to a matrix function
  const url = gateway_endpoint + "/function/matrixfunction?param=" + param;
  let res =  await testEntry(url);
  if (res) {
          return context
            .status(200)
            .succeed(res)
  } else {
      return context.fail({"result": "Error while trying to call function"});
  }
};
```

Listing 9: Chaining Function

```
package function

// Handle a serverless request
func Handle(req []byte) string {
        return "Hello From Go Serverless Function"
}
```

Listing 10: Go Function

```python
def handle(event, context):
    return "Hello From Python Serverless Function"
```

Listing 11: Python Function

```javascript
'use strict'

module.exports = async (event, context) => {
  const result = {
    'status': 'Hello From NodeJS Serverless Function'
  }

  return context
    .status(200)
    .succeed(result)
}
```

Listing 12: Javascript Function

```java
package com.openfaas.function;

import com.openfaas.model.IHandler;
import com.openfaas.model.IResponse;
import com.openfaas.model.IRequest;
import com.openfaas.model.Response;

public class Handler implements com.openfaas.model.IHandler {

    public IResponse Handle(IRequest req) {
        Response res = new Response();
            res.setBody("Hello From Java Serverless Function");

            return res;
    }
}
```

Listing 13: Java Function

Appendix B

Faas-Exp Configurations

```
experiment:
  server: gateway.openfaas.local
  port: 80
  number_of_runs: 6
  number_of_requests: 35000
  delay_between_runs: 1
  replicas:
    - 1
    - 10
    - 20
  concurrency:
    - 5
    - 10
    - 20
    - 50
  result_dir: /home/centos/result

functions:
  - name: warmfunction
    inactivity_duration: 9
    chunks_number: 6
    yaml_path: functions/warm-starts-scenarios/k8s/warmfunction.yml
    environment:
      read_timeout: 5m5s
      write_timeout: 5m5s
    api:
      uri: function/warmfunction
      data: "Hello warmfunction on k8s"
      http_method: POST
```

```
- name: gofunction
  yaml_path: functions/runtimes-scenarios/go/common/gofunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
  api:
    uri: function/gofunction
    http_method: POST

- name: javafunction
  yaml_path: functions/runtimes-scenarios/java/common/javafunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
  api:
    uri: function/javafunction
    http_method: POST

- name: nodefunction
  yaml_path: functions/runtimes-scenarios/nodejs/common/nodefunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
  api:
    uri: function/nodefunction
    http_method: POST

- name: pythonfunction
  yaml_path: functions/runtimes-scenarios/python/common/pythonfunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
  api:
    uri: function/pythonfunction
    http_method: POST

- name: iofunction
  yaml_path: functions/computation-scenarios/io/common/iofunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
  api:
    uri: function/iofunction
    http_method: POST
    data: >
      Lorem Ipsum is simply dummy text of the printing and typesetting industry
```

```
- name: matrixfunction
  yaml_path: functions/computation-scenarios/matrix/common/matrixfunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
  api:
    uri: function/matrixfunction
    param:
      min: 10
      max: 200
    http_method: POST

- name: consumerfunction
  yaml_path: functions/composite-scenarios/consumer/swarm/consumerfunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
    gateway_endpoint: http://gateway:8080
  api:
    uri: function/consumerfunction
    param:
      min: 5
      max: 150
    http_method: POST
  depends_on: matrixfunction

- name: ftpfunction
  yaml_path: functions/computation-scenarios/network/common/ftpfunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
    ftp_host: 10.0.2.207
    ftp_user: ftpuser
    ftp_password: ftppassword
  api:
    uri: function/ftpfunction
    http_method: POST
```

Listing 14: K8S Configuration

```yaml
experiment:
  server: gateway.openfaas.local
  port: 80
  number_of_runs: 6
  number_of_requests: 35000
  delay_between_runs: 1
  replicas:
    - 1
    - 10
    - 20
  concurrency:
    - 5
    - 10
    - 20
    - 50
  result_dir: /home/centos/result

functions:
  - name: warmfunction
    inactivity_duration: 9
    chunks_number: 6
    yaml_path: functions/warm-starts-scenarios/common/warmfunction.yml
    environment:
      read_timeout: 5m5s
      write_timeout: 5m5s
    api:
      uri: function/warmfunction
      data: "Hello warmfunction on swarm"
      http_method: POST

  - name: gofunction
    yaml_path: functions/runtimes-scenarios/go/common/gofunction.yml
    environment:
      read_timeout: 5m5s
      write_timeout: 5m5s
    api:
      uri: function/gofunction
      http_method: POST

  - name: javafunction
    yaml_path: functions/runtimes-scenarios/java/common/javafunction.yml
    environment:
      read_timeout: 5m5s
      write_timeout: 5m5s
    api:
```

```
    uri: function/javafunction
    http_method: POST

- name: nodefunction
  yaml_path: functions/runtimes-scenarios/nodejs/common/nodefunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
  api:
    uri: function/nodefunction
    http_method: POST

- name: pythonfunction
  yaml_path: functions/runtimes-scenarios/python/common/pythonfunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
  api:
    uri: function/pythonfunction
    http_method: POST

- name: iofunction
  yaml_path: functions/computation-scenarios/io/common/iofunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
  api:
    uri: function/iofunction
    http_method: POST
    data: >
      Lorem Ipsum is simply dummy text of the printing and typesetting industry.

- name: matrixfunction
  yaml_path: functions/computation-scenarios/matrix/common/matrixfunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
  api:
    uri: function/matrixfunction
    param:
      min: 10
      max: 200
    http_method: POST

- name: consumerfunction
  yaml_path: functions/composite-scenarios/consumer/swarm/consumerfunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
    gateway_endpoint: http://gateway:8080
  api:
    uri: function/consumerfunction
    param:
      min: 5
      max: 150
    http_method: POST
  depends_on: matrixfunction

- name: ftpfunction
  yaml_path: functions/computation-scenarios/network/common/ftpfunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
    ftp_host: 10.0.2.207
    ftp_user: ftpuser
    ftp_password: ftppassword
  api:
    uri: function/ftpfunction
    http_method: POST
```

Listing 15: Docker Swarm Configuration

```
experiment:
  server: gateway.openfaas.local
  port: 80
  number_of_runs: 6
  number_of_requests: 35000
  delay_between_runs: 1
  replicas:
    - 1
    - 10
    - 20
  concurrency:
    - 5
    - 10
    - 20
    - 50
  result_dir: /home/centos/result

functions:
  - name: gofunction
    yaml_path: functions/runtimes-scenarios/go/common/gofunction.yml
    environment:
      read_timeout: 5m5s
      write_timeout: 5m5s
    api:
      uri: function/gofunction
      http_method: POST

  - name: javafunction
    yaml_path: functions/runtimes-scenarios/java/common/javafunction.yml
    environment:
      read_timeout: 5m5s
      write_timeout: 5m5s
    api:
      uri: function/javafunction
      http_method: POST

  - name: nodefunction
    yaml_path: functions/runtimes-scenarios/nodejs/common/nodefunction.yml
    environment:
      read_timeout: 5m5s
      write_timeout: 5m5s
    api:
      uri: function/nodefunction
      http_method: POST
```

```
- name: pythonfunction
  yaml_path: functions/runtimes-scenarios/python/common/pythonfunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
  api:
    uri: function/pythonfunction
    http_method: POST

- name: iofunction
  yaml_path: functions/computation-scenarios/io/common/iofunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
  api:
    uri: function/iofunction
    http_method: POST
    data: >
      Lorem Ipsum is simply dummy text of the printing and typesetting industry.


- name: matrixfunction
  yaml_path: functions/computation-scenarios/matrix/common/matrixfunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
    write_debug: true
  api:
    uri: function/matrixfunction
    param:
      min: 10
      max: 200
    http_method: POST

- name: consumerfunction
  yaml_path: functions/composite-scenarios/consumer/nomad/consumerfunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
    gateway_endpoint: http://10.0.2.150
  api:
    uri: function/consumerfunction
    param:
      min: 5
      max: 150
    http_method: POST
  depends_on: matrixfunction

- name: ftpfunction
  yaml_path: functions/computation-scenarios/network/common/ftpfunction.yml
  environment:
    read_timeout: 5m5s
    write_timeout: 5m5s
    ftp_host: 10.0.2.205
    ftp_user: ftpuser
    ftp_password: ftppassword
  api:
    uri: function/ftpfunction
    http_method: POST
```

Listing 16: Nomad Configuration

```
version: 1.0
provider:
  name: openfaas
  gateway: http://gateway.openfaas.local
functions:
  matrixfunction:
    lang: node12
    handler: ./matrixfunction
    image: mabuaisha/matrixfunction:latest
    labels:
      com.openfaas.scale.min: 1
      com.openfaas.scale.max: 20
    secrets:
      - dockerhub
```

Listing 17: OpenFaas Function Sample Configuration