Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Sunil Kumar Mohanty

# Evaluation of Serverless Computing Frameworks Based on Kubernetes

Master's Thesis
Espoo, July 6, 2018

Supervisor: Professor Mario Di Francesco, Aalto University
Instructor: Gopika Premsankar M.Sc. (Tech.)

Aalto University
School of Science

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

| | | | |
|---|---|---|---|
| **Author:** | Sunil Kumar Mohanty | | |
| **Title:** Evaluation of Serverless Computing Frameworks Based on Kubernetes | | | |
| **Date:** | July 6, 2018 | **Pages:** | 76 |
| **Professorship:** | Mobile Computing, Services and Security | **Code:** | SCI3045 |
| **Supervisor:** | Professor Mario Di Francesco | | |
| **Instructor:** | Gopika Premsankar M.Sc. (Tech.) | | |

Recent advancements in virtualization and software architectures have led to the birth of the new paradigm of serverless computing. Serverless computing, also known as function-as-a-service, allows developers to deploy functions as computing units without worrying about the underlying infrastructure. Moreover, no resources are allocated or billed until a function is invoked. Thus, the major benefits of serverless computing are reduced developer concern about infrastructure, reduced time to market and lower cost. Currently, serverless computing is generally available through various public cloud service providers. However, there are certain bottlenecks on public cloud platforms, such as vendor lock-in, computation restrictions and regulatory restrictions. Thus, there is a growing interest to implement serverless computing on a private infrastructure. One of the preferred ways of implementing serverless computing is through the use of containers. A container-based solution allows to utilize features of existing orchestration frameworks, such as Kubernetes. This thesis discusses the implementation of serverless computing on Kubernetes. To this end, we carry out a feature evaluation of four open source serverless computing frameworks, namely Kubeless, OpenFaaS, Fission and OpenWhisk. Based on predefined criteria, we select Kubeless, Fission and OpenFaaS for further evaluation. First, we describe the developer experience on each framework. Next, we compare three different modes in which OpenFaaS functions are executed: HTTP, serializing and streaming. We evaluate the response time of function invocation and ease of monitoring and management of logs. We find that HTTP mode is the preferred mode for OpenFaaS. Finally, we evaluate the performance of the considered frameworks under different workloads. We find that Kubeless has the best performance among the three frameworks, both in terms of response time and the ratio of successful responses.

| | |
|---|---|
| **Keywords:** | docker, container, kubernetes, serverless, microservices |
| **Language:** | English |

# Acknowledgments

I would like to thank my thesis supervisor Professor Mario Di Francesco for giving me an opportunity to work on this thesis and providing valuable insights. I would like to specifically thank my instructor Gopika Premsankar who has always kept her door open to answer my unending queries. I am grateful to her for constantly guiding me through out this thesis work.

I would like to thank my parents, sisters, in-laws and friends for their constant support and motivation. Special thanks to my wife for constantly encouraging me throughout my years of studies and through the process of writing this thesis. This thesis would not have been possible without their help and support.

Thank you.

Espoo, July 6, 2018

Sunil Kumar Mohanty

# Abbreviations and Acronyms

| | |
|---|---|
| 3GPP | 3rd Generation Partnership Project |
| API | Application programming interface |
| AWS | Amazon Web Services |
| CD | Continuous Delivery |
| CDN | Content Delivery Network |
| CI | Continuous Integration |
| CLI | Command-line Interface |
| CNCF | Cloud Native Computing Foundation |
| FaaS | Function as a Service |
| GKE | Google Kubernetes Engine |
| HPA | Horizontal Pod Autoscaler |
| IaaS | Infrastructure as a Service |
| IoT | Internet of Things |
| IPC | Interprocess Communication |
| PaaS | Platform as a Service |
| QPS | Queries per second |
| REST | Representational State Transfer |
| RBAC | Role Based Access Control |
| SaaS | Software as a Service |
| UTS | Unix Time Sharing |
| VM | Virtual Machine |
| VMM | Virtual Machine Monitor |

# Contents

**Abbreviations and Acronyms** **4**

**1 Introduction** **8**
1.1 Motivation . . . 9
1.2 Contribution . . . 10
1.3 Structure of thesis . . . 11

**2 Background** **12**
2.1 Virtualization . . . 12
2.1.1 Hypervisor-based virtualization . . . 13
2.1.2 Container-based virtualization . . . 14
2.2 Docker . . . 17
2.3 Container orchestration . . . 21
2.3.1 Docker Swarm . . . 21
2.3.2 Kubernetes . . . 23
2.4 Microservices . . . 27
2.5 Serverless computing . . . 29
2.5.1 Definition . . . 30
2.5.2 Existing platforms . . . 31
2.5.3 Use cases . . . 32
2.5.4 Benefits and Challenges . . . 33

**3 Open source serverless frameworks** **35**
3.1 Evaluation criteria . . . 35
3.2 Frameworks . . . 36
3.2.1 Kubeless . . . 36
3.2.2 Apache OpenWhisk . . . 38
3.2.3 Fission . . . 40
3.2.4 OpenFaaS . . . 41
3.3 Summary and comparison . . . 43

**4 Evaluation** **46**
- 4.1 Methodology . . . 46
- 4.2 Ease of development . . . 47
- 4.3 Comparison of Watchdog modes in OpenFaaS . . . 49
  - 4.3.1 Setup . . . 49
  - 4.3.2 Results . . . 53
- 4.4 Performance of the different frameworks . . . 55
  - 4.4.1 Setup . . . 56
  - 4.4.2 Results . . . 57

**5 Conclusion** **60**

**A Fission** **70**
- A.1 Function code . . . 70
- A.2 Examples of Commands . . . 70

**B Kubeless** **71**
- B.1 Function code . . . 71
- B.2 Examples of Commands . . . 71

**C OpenFaaS** **72**
- C.1 Create a new function . . . 72
  - C.1.1 Auto generated function code . . . 72
  - C.1.2 Sample yml file for OpenFaaS functions . . . 73
- C.2 Deploy function . . . 73

**D Function Code** **75**
- D.1 Serializing Function - Classic . . . 75
- D.2 HTTP Function . . . 75
- D.3 Streaming Function . . . 76

# List of Figures

2.1 Hypervisor-based virtualization [60]. . . . . . . . . . . . . . . . 13
2.2 Container-based virtualization. . . . . . . . . . . . . . . . . . . . . 14
2.3 cgroup hierarchical structure. . . . . . . . . . . . . . . . . . . . . 16
2.4 Docker architecture [60]. . . . . . . . . . . . . . . . . . . . . . . . 18
2.5 Multiple Docker containers using the same image. . . . . . . . . 20
2.6 Docker Swarm architecture. . . . . . . . . . . . . . . . . . . . . . 21
2.7 Kubernetes architecture. . . . . . . . . . . . . . . . . . . . . . . . 24
2.8 Monoliths and microservices [54]. . . . . . . . . . . . . . . . . . . 27
2.9 Google Trends of “serverless” in last five years. . . . . . . . . . . 30
2.10 Developer control in different computing paradigms [40]. . . . 30

3.1 High-level architecture of Kubeless. . . . . . . . . . . . . . . . . 37
3.2 OpenWhisk: high level architecture [3]. . . . . . . . . . . . . . . 38
3.3 OpenWhisk system architecture [3]. . . . . . . . . . . . . . . . . 39
3.4 Fission system architecture [14]. . . . . . . . . . . . . . . . . . . 40
3.5 OpenFaaS system architecture [29]. . . . . . . . . . . . . . . . . 42
3.6 OpenFaaS: interaction between components [17]. . . . . . . . . . 43

4.1 System architecture of OpenFaaS on a Kubernetes cluster. . . 50
4.2 OpenFaaS watchdog modes [34]. . . . . . . . . . . . . . . . . . . 52
4.3 Average response time (in ms) for different OpenFaaS watchdog modes. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.4 Density of response time for different watchdog modes in OpenFaaS. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
4.5 Average response time (in milliseconds) for different serverless frameworks with 1, 5, 25 and 50 function replicas. . . . . . . . 58

# Chapter 1

# Introduction

Recent advancements in technology have been pushing software developers to generate value quickly and release new features faster than before. The time to market is also becoming a critical metric, driven primarily by the fear of competition [48]. In the past, developers spent a significant amount of time in planning and maintaining infrastructure in addition to writing code for business logic. Developers purchased bare metal servers and either installed them on-premise or leased rack space in data centers. Moreover, they had to over-provision their infrastructure to account for scalability and resilience [37]. Thus, much of the infrastructure remained unutilized for a significant portion of time. The advent of virtualization improved the utilization of the underlying hardware, resulting in significant reduction in cost [39, 62]. This development led to the rise of cloud computing and developers started treating hardware as a utility. Presently, developers no longer need to own the actual hardware and pay only for what they use. Of the several virtualization technologies available, hypervisor-based virtualization and container-based virtualization are the most popular ones [53, 64]. Container-based virtualization has become more popular in the recent years as it allows for better utilization of resources, faster provisioning and de-provisioning, and rapid scalability when compared to hypervisor-based virtualization [61]. While hypervisor-based virtualization provides *computing resources* on demand, container-based virtualization provides *applications* on demand. Advancements in container technology have made developers switch from a large monolithic architecture to an architecture based on distributed microservices [51].

Such a shift to containers and microservices architecture has given birth to a new paradigm known as *serverless computing* [45]. In serverless computing (also known as *function-as-a-service*), functions are written and deployed to a platform without developers worrying about the underlying infrastruc-

ture. The latter is generally managed by a third-party service provider or a different team (when using a private cloud). It is important to note that serverless computing does not mean the absence of servers. The term 'serverless' is a misnomer and a marketing jargon made popular by the Amazon Web Services (AWS) Lambda[1] service. The implementation still happens on real servers, but developers are relieved of installing and managing the infrastructure. The development team can then focus on the business logic and deploy the functions as soon as they are ready. These functions are completely stateless, allowing them to scale rapidly and independent from each other. As a consequence, serverless computing has made developers change the way they design, develop and deploy modern software [37].

## 1.1 Motivation

The main motivation behind the rapid adoption of serverless computing are: decreased concern for developers, reduced time to market, billing for code execution instead of resource usage, and reduced effort in application management and operations. Although utilizing serverless offerings of public cloud service providers is very easy and convenient, there are several scenarios where developers prefer their private infrastructure. Some of the major reasons are as follows. Certain organizations would like to use their own infrastructure for regulatory reasons or also to utilize existing private infrastructure. A serverless environment allows them to bill individual business units based on execution time, thus creating a simple and cheaper cost model within the organization. Moreover, each public cloud provider has its own implementation of serverless computing, forcing developers to write their code in a certain way. Hence, one of the major risks in adopting a public serverless platform is vendor lock-in. Each public cloud provider also has limitations in terms of languages supported, maximum execution duration, maximum concurrent executions and so on. All these bottlenecks can be addressed if serverless computing is realized in a private cloud environment, thus giving an organization complete control over the infrastructure.

Edge computing, specifically data analytics at the edge, is particularly suited for serverless computing [67]. Presently, Internet of Things (IoT) sensors produce a large volume of data that has to be analyzed to extract useful information. A private serverless infrastructure can be created on the IoT gateways to process the data with very low latencies (as they are located close to the devices generating the data). This is all the more beneficial when the

[1] https://aws.amazon.com/lambda/

IoT gateways do not have an Internet connection to connect to cloud services. With a serverless environment, consumers can deploy their functions and be billed for only the execution time at the edge. Here, function-as-a-service can be combined with gateway as a service [65]. Such a model will simplify the programming of IoT devices and improve the utilization of resources at the gateways.

The reasons described above have motivated us to implement serverless computing on a private infrastructure. Indeed, serverless computing can be implemented in a private cloud through containers [63]. There are many open source tools available today which have made orchestration (i.e., the deployment and scaling) of containers easy. One of the most popular open source container orchestration tools is Kubernetes[2], which is developed by Google. The goal of this thesis is to implement and evaluate serverless computing in a private cloud by using Kubernetes as a container orchestration tool. Specifically, we aim to evaluate existing open source serverless computing frameworks in terms of their features and performance.

## 1.2 Contribution

The contributions of this work are the following.

- Establishing the feasibility of setting up a serverless platform on a private cloud using Kubernetes as a container orchestration tool
- Evaluating the features of existing open source serverless frameworks, namely, OpenFaaS, Kubeless, Fission and OpenWhisk
- Implementing a serverless environment on virtual servers by using OpenFaaS, Kubeless and Fission
- Evaluating the serverless platforms in terms of ease of development and monitoring of the functions; in particular, the steps needed to build and deploy functions on each considered framework
- Evaluating the performance (in terms of response time and ratio of successful responses) of the considered frameworks under different workloads

[2]https://kubernetes.io/

## 1.3 Structure of thesis

The rest of this thesis is structured as follows. Chapter 2 introduces the relevant background about virtualization, containers, Kubernetes, microservices and serverless computing. Chapter 3 discusses several open source solutions to implement serverless computing. Chapter 4 describes the implementation of serverless computing on a Kubernetes cluster by using OpenFaaS, Kubeless and Fission. It also evaluates the performance of the serverless environment with each of these frameworks. Finally, Chapter 5 provides some concluding remarks and directions for future work.

# Chapter 2

# Background

The chapter provides background information on the technologies and concepts relevant to this thesis. Section 2.1 introduces the concept of virtualization and describes hypervisor and container-based virtualization. Section 2.2 describes the features of Docker, a popular software tool for Linux containers. Section 2.3 discusses two popular container orchestration frameworks, Kubernetes and Docker Swarm. Section 2.4 presents the mircroservices architecture, one of the driving forces behind serverless computing. Finally, Section 2.5 describes the concept of serverless computing, its benefits and disadvantages and highlights certain use cases for serverless computing.

## 2.1 Virtualization

Virtualization lies at the heart of serverless computing. Hence, it is important to clearly introduce virtualization to better understand serverless computing. Although virtualization has gained rapid popularity in the last few years, its origin can be traced to 1960's when IBM introduced the idea with the M44/44X system [49]. Virtualization enables developers to run multiple virtualized instances on top of a server. The instances run in complete isolation and also provide rapid scalability, better utilization of computing resources and reduction in cost [39, 62]. The two most popular virtualization technologies are hypervisor-based virtualization and container-based virtualization (also known as operating system-level virtualization). We describe these technologies next.

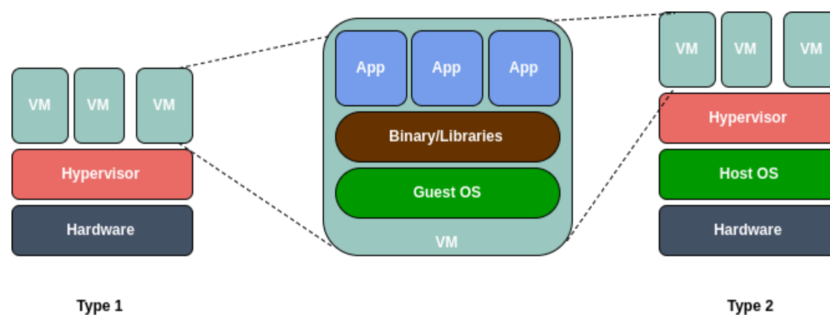



Figure 2.1: Hypervisor-based virtualization [60].

### 2.1.1 Hypervisor-based virtualization

In the words of Popek and Goldberg [69], a virtual machine (VM) can be defined as *"an efficient, isolated duplicate of the real machine"*. Over the last decade, hypervisor-based virtualization has been a popular method for implementing VMs. This approach relies on a software called a hypervisor or virtual machine monitor (VMM) that lies between the hardware and VMs. It can host multiple VMs on top of it. The VMM has the following three characteristics [69]: provides environments for programs which are identical to the original machine had the program been run directly there; ensures minimal performance degradation for programs running in these environments; the VMM has complete control over the resources on the host system.

Hypervisor-based virtualization can be further divided into two types (Figure 2.1): Type 1, which is native or bare metal and Type 2, which is hosted [57, 73]. A **Type 1** hypervisor architecture runs the hypervisor directly on top of the underlying host's hardware. The VMs run on top of the hypervisor. Here all the necessary scheduling and resource allocation are done by the hypervisor. Some of the notable hypervisors based on this architecture are Xen [42], Oracle VM, VMware ESX [66], Microsoft Hyper-V [78]. In a **Type 2** hypervisor architecture, the hypervisor runs on top of the host operating system (OS). The VMs run on top of the hypervisor. Here, the hypervisor relies on the host OS for processor scheduling and resource allocation. Some of the notable hypervisors based on this architecture are Oracle VirtualBox, VMWare Workstation [77], Microsoft Virtual PC [58], QEMU [43], and Parallels[1].

The major benefit of hypervisor-based virtualization is that it allows users to run multiple VMs in isolation on a single machine. This allows developers

[1]http://www.parallels.com

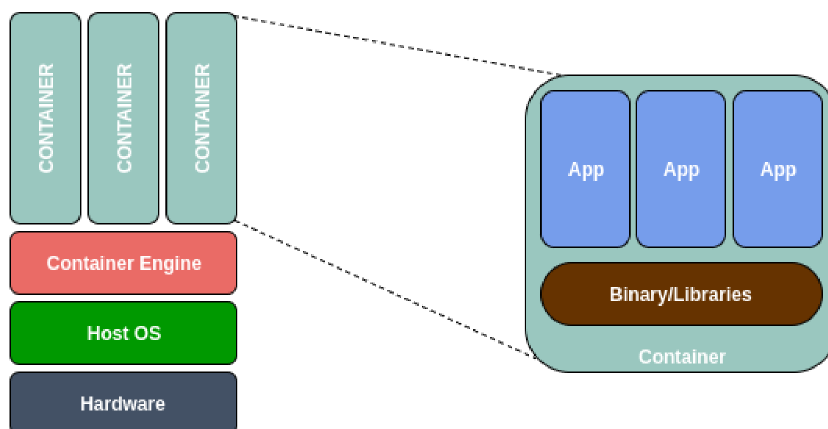



Figure 2.2: Container-based virtualization.

to create multiple environments with different hardwares and OS on a single hardware. For example, a VM can be easily configured with 1 GB of memory and 1 CPU on a host machine having 32 GB memory and 4 CPUs. The same VM can then be reconfigured to have 4 GB of memory and 2 CPUs without making any actual hardware changes, but only some configuration changes. Developers can save a VM as an image and port this image to different hardware or data centers. This gives developers a seamless experience from development to testing to production. Furthermore, cloud service providers have created interfaces which allow the creation of the VMs through APIs. This has allowed developers to scale applications based on demand which resulted in the initial wave of cloud computing.

Despite the many benefits, hypervisor-based virtualization has its limitations. Firstly, there is some performance deterioration in VMs as compared to native machines [42]. Moreover, VMs can be slow as they still need to be booted up like a normal OS, resulting in long start up time.

### 2.1.2 Container-based virtualization

Containers, also known as operating system (OS)-level virtualization, are a lightweight alternative to hypervisor-based virtualization. Containers create multiple, isolated userspace instances on top of the same OS kernel [80]. Thus containers provide an abstraction on top of the OS kernel allowing multiple guest processes to run within a container in isolation from other containers. As shown in Figure 2.2, each container behaves as an independent OS without the need for an intermediate layer such as a hypervisor.

There are multiple solutions for OS-level virtualization. Linux-VServer[2] is one of the oldest implementations. It uses the `chroot()` barrier to prevent unauthorized modification of the file system [76, 80]. OpenVZ[3] relies on kernel namespaces, allocating a PID namespace to each of the containers to isolate them [76, 80]. OpenVZ relies on a modified Linux kernel and several user-level tools [46]. Virtuozzo[4] is another implementation of Linux container technology and is based on OpenVZ. LXC[5] is an implementation of containerization which uses Linux kernel functionalities already implemented in the upstream Linux kernel [72]. Moreover, the use of Linux Security Modules by LXC makes it stand out [72]. In this thesis, we use the LXC implementation of containers to achieve our goal of creating a serverless environment. Next, we describe the key enabling Linux kernel functionalities of LXC.

### Control groups

Control groups[6], also called as cgroups, enable LXC to control resources. These resources include memory, CPU, block I/O, devices and traffic controller [72]. It provides a mechanism for aggregating sets of processes into hierarchical groups and controls the allocation of resources per cgroup [80]. Each groups inherits from its parent. As shown in Figure 2.3, each process belongs to a node in the hierarchy which is arranged as a tree and each node can have multiple processes sharing the same set of resources. Cgroups associate these sets of processes to one or more subsystems. Subsystems are controllers which control access to the underlying resource. For example, the memory controller keeps track of pages used by each group. Each cgroup also has a virtual file system associated with it. Cgroups are configurable allowing for dynamic allocation of resources. Thus, cgroups are responsible for accounting and limiting how many resources a set of processes in a cgroup can access.

### Namespaces

Namespaces[7] allow different processes to have their own view of the system resources and are used by containers to provide resource isolation. Changes made to the resource in a namespace are only visible within the same names-

[2]http://linux-vserver.org
[3]http://www.openvz.org
[4]http://www.virtuozzo.com
[5]https://linuxcontainers.org
[6]https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt
[7]http://man7.org/linux/man-pages/man7/namespaces.7.html

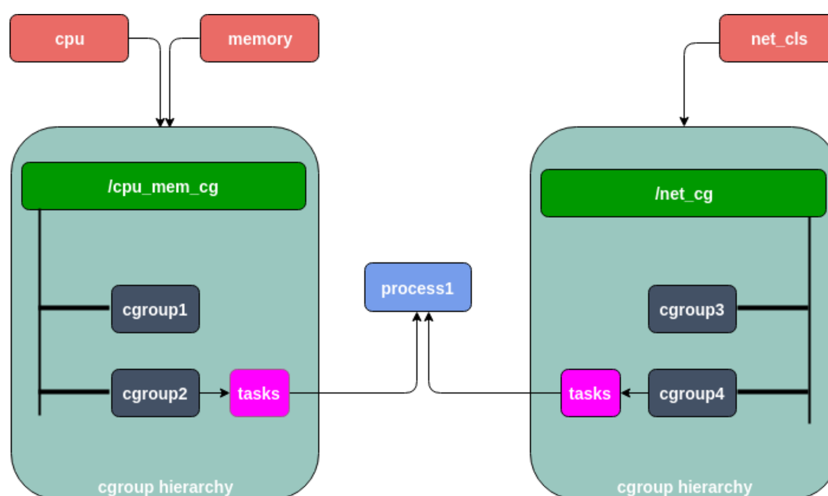



Figure 2.3: cgroup hierarchical structure.

pace. This makes the process have their own isolated instance of global resources. The Linux kernel has seven different namespaces:

- *Cgroup namespaces* are responsible for virtualizing the view of process's cgroup. Each cgroup namespace has its own cgroup root directories.
- *Interprocess Communication (IPC) namespaces* isolate IPC resources, such as System V IPC objects and Posix message queues. Each IPC namespace has its own set of IPC resources and their visibility is limited to the namespace to which they belong.
- *Network namespaces* isolate the networking resources, such as networking devices, IPv4 and IPv6 protocol stacks, routing tables, firewalls, and so on. A physical device is visible in only one network namespace. Networking between namespaces can be done by creating tunnels using a virtual Ethernet interface.
- *Mount namespaces*[8] isolate the list of mount points in each namespace instance, thereby restricting the processes in each mount namespace instance to see distinct directory hierarchies.
- *PID namespaces*[9] isolate the processID, resulting in processes in different PID namespaces having the same PID number. This functionality

[8]http://man7.org/linux/man-pages/man7/mount_namespaces.7.html
[9]http://man7.org/linux/man-pages/man7/pid_namespaces.7.html

allows a container to suspend or resume a set of processes within it and then migrate the container to a new host, retaining the same PID number.

- *User namespaces*[10] isolate security-related identifiers and attributes, such as user ids, group ids, root directory, keys and capabilities. For example, a process can have full privileges inside a user namespace, but at the same time be unprivileged outside the user namespace.
- *Unix Time Sharing (UTS) namespaces* isolate two system identifiers: the hostname and the domain name. This allows a single system to appear to have a different host name and domain name to different processes.

Containers have been challenging hypervisor-based virtualization for adoption by cloud computing providers [64]. There are many reasons for this. Unlike VMs, containers share the same Linux kernel. Hence, container images are not full-fledged operating systems like VMs. This makes containers very light-weight, scalable and easy to port. Containers are faster to boot when compared to VMs [52]. In fact, the performance of containers is near native in comparison to hypervisor-based virtualization [64]. Since containers have the ability to return unused resources to the host machine or other containers, resource utilization is better than hypervisor-based virtualization wherein resources are locked for each VM [80]. However, since containers on the same host share the same Linux kernel, they provide less isolation as compared to VMs.

## 2.2 Docker

Docker[11] is an open source platform for building, shipping and deploying containers. It uses the resource isolation features provided by the Linux kernel, such as cgroups and namespaces [38]. Docker-based applications can be deployed faster than traditional non-containerized applications. Moreover, Docker allows developers to configure many infrastructure components, such as memory, CPU and networking, through a Dockerfile (described later) or the command line. This allows developers to manage infrastructure similar to how they manage applications. Ultimately, it helps developers in reducing the time to market [31]. Docker provides a platform to run almost any

[10] http://man7.org/linux/man-pages/man7/user_namespaces.7.html
[11] https://docs.docker.com/

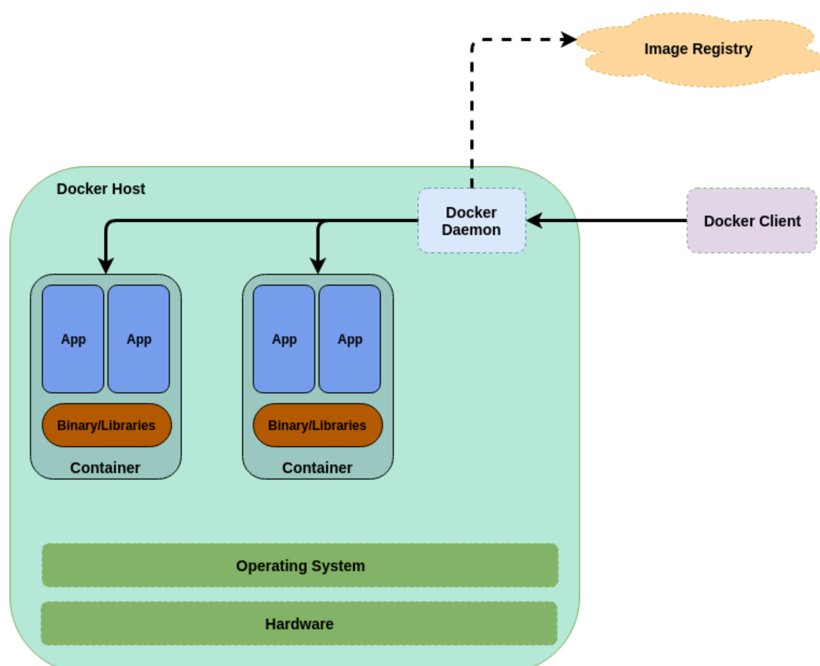



Figure 2.4: Docker architecture [60].

application in a secure and isolated manner in a container. This security and isolation allows Docker to run multiple containers simultaneously on top of a single host. Docker provides all the tooling required to manage the containers [70]. Developers can develop applications and its components by using containers. These containers can then be packaged, distributed and deployed by using Docker. The lightweight nature of Docker further helps the distribution.

### Docker architecture

Docker consists of Docker Engine[12] which is responsible for creating and running Docker containers. Docker engine is based on a client-server architecture. Users interact with the Docker engine by using the Docker client. Figure 2.4 illustrates the Docker architecture. We describe the components of the Docker platform next.

- The *Docker client* is the primary interface that is used by Docker users to interact with the Docker platform. It is a command line interface

[12] https://docs.docker.com/engine/

by which the users send commands. The client then sends these commands to the Docker daemon which executes them. The client uses a Representational State Transfer (REST) Application Programming Interface (API) to interact with the Docker daemon. The client and daemon need not be on the same host.

- The *Docker daemon* is responsible for managing Docker containers on the host system. It runs on the host machine and waits for Docker API requests from the Docker client. Users cannot interact with the Docker daemon directly.

- *Docker images* are read-only templates with the source code for creating and running containers. These are the primary building blocks of Docker containers. Every Docker image contains libraries and binaries that are necessary to build applications. Developers can build their own image or download images built by other developers. To create their own image, developers write instructions in a Dockerfile (described next). Each step in the Dockerfile creates a layer in the image and the layers are combined by using a file system called Union file system (UnionFS). Whenever a change is made to the image, a new layer containing only the changes is added on top of the existing layers. This allows for the layers to be used as a cache which can be reused. The layers are read only in nature. Whenever developers make any small change to their code, Docker uses the image from the cache and create a new layer on top of the image with just the necessary changes. This also makes the process for rebuilding images fast [38].

- *Dockerfile*[13] is a text file that contains all the information needed to build a Docker image. User can use the `docker build` command to build an image from a Dockerfile. The Dockerfile is composed of various instructions which are based on a simple syntax and are executed in sequence. Docker goes through the Dockerfile and executes each instruction in the order specified. While executing the instructions, Docker checks if it can reuse an image from the cache instead of creating a new duplicate image. A Dockerfile must start with a 'FROM' instruction, which specifies the base image from which to build the image. The base image can also be SCRATCH, which instructs Docker to start with an empty filesystem as the base. The subsequent lines in the Dockerfile adds, deletes and modifies files or configurations.

[13] https://docs.docker.com/engine/reference/builder

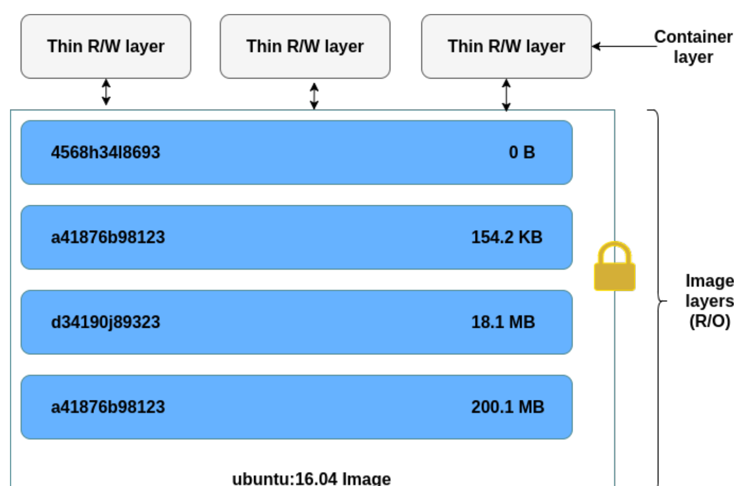



Figure 2.5: Multiple Docker containers using the same image.

- *Docker registries* are used to store images. They can be either public or private. Developers can build images and store them in these registries to make the distribution of the images easy. Users can then access and download Docker images from these registries by using Docker client. These registries behave similar to source code repositories [59]. One of the most popular and largest public Docker registries is Docker Hub[14]. Developers can also create private repositories and thus control access to the images.

- *Docker containers* are the runnable instances of the Docker images. They are created, started, stopped and deleted through the Docker client. Multiple instances of containers can be created from the same Docker image on the same host. Each of these containers run in complete isolation of each other. Users can attach persistent storage and network to the containers. As shown in Figure 2.5, when a container is created, a new writable layer called the "container layer" is created on top of the underlying images. Changes made in the running container are written to this container layer (unless the changes are targeted towards the persistent storage). When multiple containers are created from the same image, they create their own writable container layer [1]. When a container is stopped, the changes made in the writable container layer are lost.

[14] https://hub.docker.com

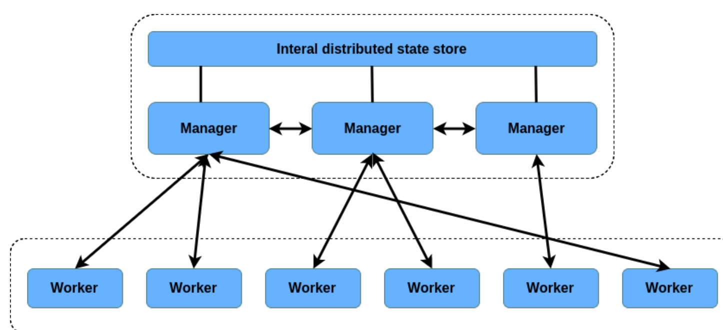



Figure 2.6: Docker Swarm architecture.

## 2.3 Container orchestration

As discussed earlier, containers, specifically Docker, make it easy to package, port and deploy applications. These features allow distributed systems to scale up and down easily. Moreover, in a distributed architecture, it is critical that multiple containers can interact among themselves. As the number of containers grow, it becomes very important to automate the whole container management process. This automation is achieved by a container orchestration framework. The primary jobs of a container orchestration framework are to provision hosts, start containers, stop containers, provide resilience, link containers, scale containers, update containers, expose containers to the external world, and so on. Of the many orchestration frameworks available in the market, the most popular are Docker Swarm and Kubernetes.

### 2.3.1 Docker Swarm

Docker Swarm[15] is a container orchestration tool which provides native Docker clustering and scheduling capabilities. The swarm mode is built into Docker Engine v1.12 and later. Swarm commands are also executed through the Docker command line interface and Docker Engine API. Hence, a swarm can be set up with minimal configuration and with very few commands. This makes swarm one of the simpler docker orchestration frameworks available. Some of the key components of Docker swarm are as follows:

- *Nodes.* A swarm consists of multiple Docker hosts, each of them running in swarm mode. The hosts are referred to as nodes. As shown

[15]https://docs.docker.com/engine/swarm/key-concepts/

in Figure 2.6, the nodes act as a manager, a worker or both. *Managers* are responsible for carrying out cluster management tasks, such as maintaining cluster state, scheduling services, and serving HTTP API endpoints for swarm mode. When an application is ready to be deployed to the Docker swarm, the service definition is submitted to the manager node. The manager node then dispatches tasks to the worker nodes. A swarm can also have multiple manager nodes for fault tolerance. When there are multiple manager nodes, a single *leader* is elected to carry out the swarm management tasks. *Worker* nodes are responsible for executing tasks received from manager nodes. Manager nodes can also act as a worker node and take on the responsibility of executing containers. By default, all managers are workers. A scheduler running on the leader decides which tasks are assigned to a worker. A worker can be in any of the three states, *Active*, *Pause* and *Drain.* A worker in the active state can be assigned to run tasks. If a node is in pause state, new containers cannot be scheduled on it although existing containers will continue to run. If a node is in drain state, no new containers can be scheduled on it and the existing containers are rescheduled on a worker in active state. Node also can have label meta data which can be used as a criteria for scheduling containers on them. For example, special containers needing fast storage can run on nodes labeled as *SSD* [18].

- *Service.* Swarm services are used to deploy containers on a swarm. The service definition contains the state of the service (application). Additionally, it contains other information such as the image name and tag, number of containers, ports to be exposed (if any), node placement preference, and if the service should start automatically when Docker starts. The node manager parses the service definition and maintains the state accordingly. For instance, if the service definition states that there should be 5 containers running at all times, the manager ensures this state is reached. Thus, if a container shuts down, it is the responsibility of the manager to start a new container. Services can be deployed in two modes, *global* and *replicated.* In the global deployment mode, a task is scheduled to run in every available node of the cluster. In case of replicated deployment mode, services define the total count of tasks, and the scheduler distributes the same on various workers taking into consideration any special placement criteria mentioned in the service definition [19].

- *Task.* Tasks in the swarm are execution units which get assigned to

a node and run on the node till completion. Their state is declared in the service and the swarm realizes the desired state by scheduling tasks [19]. For example, if a service is defined to have three instances of a specific application, the scheduler creates three tasks. Each of these tasks is realized through exactly one container. Each task goes through various states[16], such as new, pending, assigned, accepted, preparing, starting, running, complete, failed, shutdown, rejected and orphaned.

### 2.3.2 Kubernetes

Google has long been running containers in its data center [44]. Almost all the applications at Google are run on containers. These containers were managed through an internal container cluster management system called Borg. Once many external developers started getting interested in container technology and Google developed its public cloud infrastructure, it developed a new container management system called Kubernetes[17] based on Borg [44, 79]. Kubernetes is developed as an open-source product[18]. Kubernetes can be used to deploy, manage and scale containerized applications [35]. Some of the key components of Kubernetes are as follows.

- *Pods*[19] are groups of one or more containers. They are the basic building blocks in Kubernetes. All containers in a pod share the same set of resources, such as storage, IP address, port space. All the containers are tightly coupled, i.e., they are always co-located and co-scheduled. Kubernetes assigns a unique IP address to a pod. Inside a pod, the containers communicate among themselves via *localhost*. Containers in two different pods communicate via the pod IP addresses. This makes the pod behave like a VM. Kubernetes schedules and orchestrates pods and not individual containers. This allows developers to develop independent components and package them into a pod. Kubernetes sees a pods as single working units and scales them accordingly. Pods lack durability and can be evicted at any point of time. Hence, it is important for developers to keep their pods stateless.

- *Master and nodes.* Kubernetes components can be categorized as master and node components. Nodes are worker machines (such as a VM or physical machine) which run the pods and are managed by the master

[16]https://docs.docker.com/engine/swarm/how-swarm-mode-works/swarm-task-states/
[17]http://kubernetes.io/
[18]https://github.com/kubernetes/kubernetes
[19]https://kubernetes.io/docs/concepts/workloads/pods/pod/

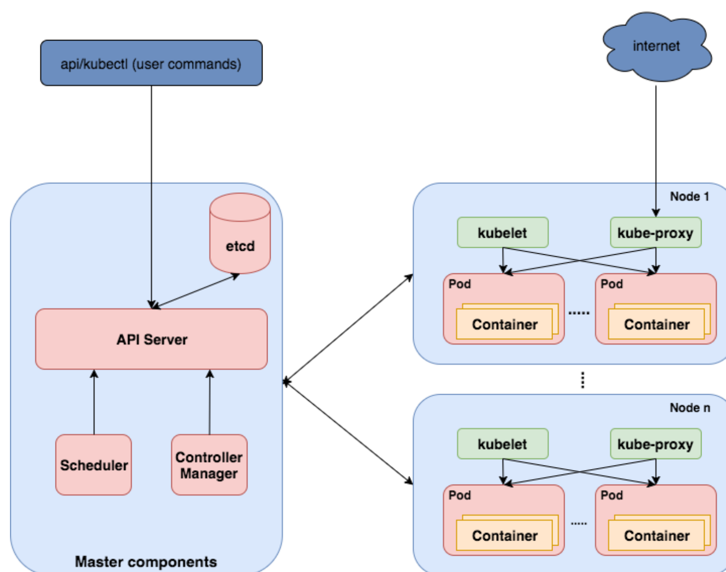



Figure 2.7: Kubernetes architecture.

components. Previously, nodes used to be called minions. Although master components can be run on any node, they are typically started on a single machine, referred to as the master. Nodes and masters are collectively known as a *cluster*. Master components provide the control plane layer in Kubernetes. They are responsible for managing containers and respond to various cluster events. Figure 2.7 shows the master components, such as the API Server, controller manager, scheduler, etcd, cloud controller manager and add-ons [27]. They are described in detail below.

- The *API server* exposes the Kubernetes API through which all communications in the cluster are done. It is also used by developers to connect to Kubernetes cluster and manipulate Kubernetes object. The APIs are versioned and are designed to scale horizontally. Developers can communicate with the API server by using REST API calls or through command line tools, such as *kubectl*[20] and *kubeadm*[21], which, in turn, use the API server. Moreover, all communication between the master and node components are handled by the API server [71].

[20]https://kubernetes.io/docs/reference/kubectl/overview/
[21]https://kubernetes.io/docs/admin/kubeadm/

- The *controller manager* runs controllers. Controllers are control loops that run on the master, check the state of the cluster through the API server and move the current state of the Kubernetes object to the desired state. There are multiple controllers running in a master. Some of the examples of controllers are replication controller, endpoint controller, node controller and service account controller. The replication controller is responsible for maintaining the desired number of pods. The endpoint controller joins the services and pods. The node controller keeps track of the availability of nodes. Finally, the service account controller creates default accounts and API access tokens [27].
- A *scheduler* is responsible for placing pods to appropriate nodes. This is done by taking into account various factors, including the current resource utilization and availability on a node and constraints specified by pods. The scheduler may evict a pod from a node based on the overall state of the cluster [27].
- *etcd*[22] is a distributed, lightweight, consistent and highly available key-value data store. It is used to store the configuration and state information of the Kubernetes cluster. It is a open source project and part of the CoreOS[23] project and uses the Raft consensus algorithm [68] to maintain high availability.
- The *cloud controller manager*[24] is a component introduced in Kubernetes v1.6 as an alpha release [26]. It is a cloud specific control loop which runs alongside other master components. Various cloud providers abstract provider-specific code into the cloud specific controller manager so as to follow an independent development lifecycle. The cloud controller manager allows cloud provides to run cloud-specific code alongside the Kubernetes controller. Some of the popular public cloud vendors, such as Amazon Web Services, Azure, Digital Ocean, Google Compute Engine and Oracle have implemented cloud controller managers for their respective cloud platforms.
- *Add-ons* are pods and services that extend the functionality of Kubernetes. An add-on manager is responsible for creating and managing the add-ons. An example of an add-on is a Web UI dashboard[25] for managing Kubernetes objects.

[22] https://github.com/coreos/etcd
[23] https://coreos.com/
[24] https://kubernetes.io/docs/concepts/architecture/cloud-controller/
[25] https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

Node components provide the run time environment for the pods. Figure 2.7 illustrates that node components run on every node. The main node components are kubelet and kube-proxy [27].

- *Kubelet*[26] is the most important node component with the responsibility to manage pods and containers in the nodes. It responds to instructions received from the master to create, monitor and destroy containers running on the node [71]. It also reports the status of nodes to the cluster. The kubelet is mainly dependent on PodSpec which is a document in YAML or JSON format that describes the pod. This document is generally served to the kubelet by the API server. It also can be an HTTP endpoint or a file path which the kubelet periodically checks for updates.
- *Kube-proxy*[27] is a network proxy that maps the individual containers to a service and provides load balancing. These proxies run on all nodes. They do not understand HTTP and can do simple TCP and UDP stream forwarding or do round robin load balancing across a set of backends.

- *Replication Controller*[28]. To handle huge load and for fault tolerance, multiple instances of the same pod need to run at the same time. Each copy of the pod is called a *replica*. These replicas are managed by the *replication controller* based on a set of rules defined in a template known as *pod template*. Based on this template, the replication controller ensures that a certain number of pods are always running in the cluster. The controller will create or delete replicas as the need may be [71]. Killing a replication controller does not kill the pods managed by it. Although it is possible to create pods without assigning it to a replication controller, it is advised not to do so (even if only a single replica of a pod is to be run). This is because pods may terminate or get destroyed unexpectedly. Attaching them to a replication controller ensures that a specified number of pods are always running in the cluster.

- *Services*[29]. While the replication controller ensures that the desired number of pods are always running in the cluster, the services make sure that the pods are accessible by the users. As discussed earlier,

[26]https://kubernetes.io/docs/reference/generated/kubelet/
[27]https://kubernetes.io/docs/reference/generated/kube-proxy/
[28]https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/
[29]https://kubernetes.io/docs/concepts/services-networking/service

pods have IP addresses assigned to them. However, as the replication controller can create or evict pods, there is no guarantee that a pod will retain its IP address. Changes in IP addresses make it difficult for users or applications to connect to the pods. Kubernetes solves this issue with the use of *services*. A Kubernetes service is an abstraction of a set of pods and defines a policy describing how to access them. The pods are normally discovered by using the label selector[30]. However, services can also be used without label selectors by mapping the service to a specific endpoint (in the form of an IP address or DNS name). Services are REST objects and are assigned IP address. Kubernetes also offers an endpoints object which is updated whenever pods linked in a service changes. When a service is created, it also creates a corresponding endpoint object which contains details of the pod based on the label selector mentioned in the service template.

## 2.4 Microservices

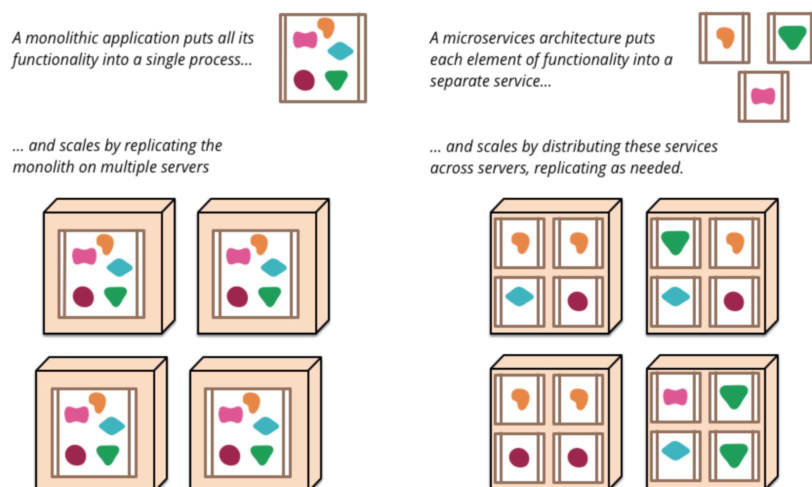



Figure 2.8: Monoliths and microservices [54].

The microservice architecture has been getting a lot of traction among software developers. Many leading companies, such as Amazon and Netflix, have adopted the microservice architecture in their products, which has further fueled the interest of software developers [36]. Figure 2.8 contrasts

[30]https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

the traditional monolithic software architecture to microservices. In monolithic applications, the complete application logic is encompassed in a single unit. The application may consist of several libraries and components but is deployed as a single unit [50]. On the other hand, in the microservices paradigm, an application is composed of many small, independent services. These services communicate by a lightweight mechanism, often an HTTP API [54, 56]. Monolithic applications can be scaled by deploying multiple copies of the same application. However, the biggest drawback of a monolithic application is that it is difficult to understand and modify. This becomes more apparent as the size and complexity of the application grows. Even a small change in the application requires the complete application to be rebuilt and redeployed. Since the monolithic application's code base is generally huge, the build time is generally long. Moreover, it follows the principle of "one size fits all" for the different modules which could compromise resource requirements. For example, certain modules may need more memory whereas the others may need more CPU. However, all modules are deployed together in one environment. Additionally, the architecture bears the burnt of technology lock-in, forcing the developer to use a single language and framework [51]. On the other hand, in the case of microservices, each service can be designed, developed, built, deployed and scaled independently. Each service can have its own programming language and framework. Thus, developers can chose the most suitable environment for each service. Microservices has been identified as a solution to efficiently build and manage complex software systems [75].

Despite the long list of benefits provided by the microservice architecture, it has many challenges. Since the architecture is distributed in nature, it can be difficult to identify and isolate errors [59]. The distributed nature also brings in dependence on the network for inter-service communication. The assumption of a reliable, homogeneous and secure network are some of the fallacies of distributed computing [21]. Moreover, the dependence on network communication can have an adverse impact on performance as network calls are slower than in-memory calls [51]. In the microservice architecture, as an application becomes bigger, so does the number of services. This increase in the number of services can make an application more fault prone [51]. Microservices lead to smaller and independent services resulting in improved service level testability. However, integration testing can become complex, especially when the system consists of a large number of services [51, 59].

## 2.5 Serverless computing

Traditionally, apart from writing code, software developers had to manage the operational concerns of deploying their application to production. When applications are deployed to a bare metal server, developers have to manage the physical servers, networking, storage, load balancers, operating systems, run times and the application itself. In this scenario, a physical server is the unit of scale and it could take days to order and deploy a new server. Thus, over-provisioning is necessary to meet peak traffic demand. The advent of virtualization automated many of these concerns and with a few mouse clicks new virtual machines could be provisioned. However, developers still need to manage the network, load balancers, operating systems and runtimes. Furthermore, scaling of the VMs takes a few minutes. The subsequent evolution of virtualization technology, i.e., containers, abstracted away many operating concerns from developers. Container-based applications could be scaled in a matter of seconds. This also spurred the growth of the microservice architecture, wherein large applications are decomposed into multiple independent container-based services. A natural extension of this approach is to decompose applications into multiple independent *stateless functions*. This is known as the *serverless computing* paradigm. Here, almost all operating concerns are abstracted away, allowing the developers to just write code and deploy their stateless functions on the serverless platform. The platform takes care of executing the functions, storage, server, operating system, container infrastructure, networking, scalability and fault tolerance. Additionally, unlike for bare metal or virtualized instances, the serverless platform takes care of scaling. Thus, serverless computing is a higher form of managed services wherein the smallest unit of computing is a function. Hence, serverless computing is also sometime referred to as *function-as-a-service (FaaS)*. Computing resources required to execute a function are provisioned only on demand and thus, the containers need not be running at all times. The functions can be kept active (hot) for a certain duration to improve performance but they are not perpetually active.

Serverless computing is actually a marketing jargon and it does not mean the complete absence of servers. It simply implies that developers do not have to manage the underlying servers and network infrastructure. When serverless computing is provided by cloud providers, customers are billed only for the duration of code execution. Serverless computing has been increasing in popularity recently. Figure 2.9 shows the increased interest on “serverless” in the last 2 years as reported by Google Trends.

Figure 2.9: Google Trends of "serverless" in last five years.

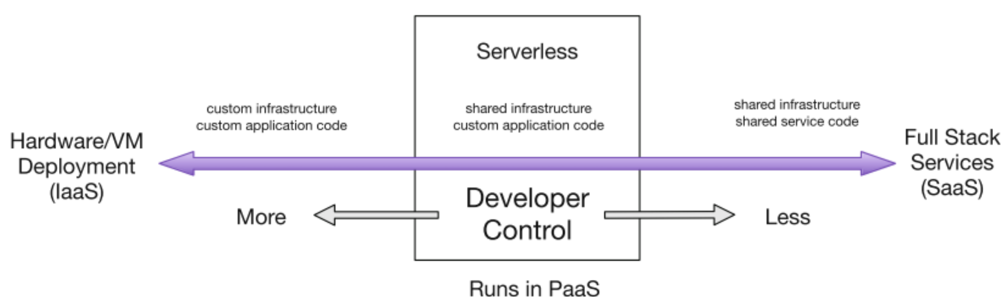



Figure 2.10: Developer control in different computing paradigms [40].

### 2.5.1 Definition

Serverless computing is still an evolving computing paradigm and does not have a clear definition which is accepted by the industry or academia. Hence, we review some definitions from the literature. Baldini et al. [40] define serverless computing based on the level of control a developer has over infrastructure. As illustrated in Figure 2.10, the authors have placed serverless computing between IaaS (Infrastructure as a Service) and SaaS (Software as a Service). In IaaS, the developer is responsible for provisioning and maintaining both the infrastructure and application. They have to customize how applications are deployed and scaled. In SaaS, the developers do not have any knowledge or control over the infrastructure. They also have little flexibility in terms of the packages and components provided by the SaaS platform. However, in a serverless computing paradigm, the developers have complete control over the code that is deployed. The serverless platform takes care of the operational aspects of the server, network, load balancing and scaling. Developers have to write stateless code and deploy the functions to the plat-

form. The platform may run zero to thousands of instances of the function based on demand and the developers are billed only for the duration their functions run. Castro et al. [45] describe serverless computing as a cloud-native paradigm which is suitable for short duration stateless functions. The functions are mostly event driven in nature. The platform can respond to bursty workloads by scaling out the functions and scaling in when there is fall in demand. Developers relinquish infrastructure design, quality of service, scaling and fault tolerance to the platform provider. The Cloud Native Computing Foundation (CNCF) defines serverless computing as a concept of building and running applications where server provisioning, maintenance, updates, scaling and capacity planning are abstracted away for developers leaving them to focus on writing code. It also identifies that the serverless platforms provide a big advantage to the consumers by not charging them when their code is idle [5].

### 2.5.2 Existing platforms

All the major cloud service providers have a serverless computing platform in their offering. Amazon Web Services (AWS) has AWS Lambda that lets developer run code without provisioning any servers. They also provide AWS Greengrass[31] to run AWS lambda functions on edge devices. AWS Step Functions[32] provide a state machine for lambda functions, allowing lambda functions to be run in step-by-step manner. Microsoft Azure provides Azure Functions[33], a serverless computing platform which allows developers to run their code as functions on Azure infrastructure. IBM also has a serverless computing platform called IBM Cloud Functions[34] which is built on top of Apache OpenWhisk[35] (described in detail in the next chapter). Google Cloud Functions[36] allows developers to run Node.js code as event-driven serverless functions.

All the above serverless offerings require the functions to be written or deployed in a certain way, resulting in vendor lock-in. Thus, there are several open source FaaS frameworks which allow to run serverless computing on private infrastructure, thereby avoiding any form of vendor lock-in. Some of the

[31] https://aws.amazon.com/greengrass
[32] https://aws.amazon.com/step-functions/
[33] https://azure.microsoft.com/en-us/services/functions/
[34] https://www.ibm.com/cloud/functions
[35] https://openwhisk.apache.org/
[36] https://cloud.google.com/functions/

popular ones are Kubeless[37], OpenFaaS[38], Fission[39] and Apache OpenWhisk. We describe these in detail in the next chapter.

### 2.5.3 Use cases

Serverless computing is ideal for workloads which have sporadic demands. They are ideal for workloads that are short, asynchronous or event-driven and concurrent work loads [5]. We identify the following use-cases as particularly suited for serverless computing.

- *Database triggers.* Event driven computing was one of the main drivers behind serverless computing. Functions as a service can be used to respond to changes in database, such as insert, update and delete operations. For instance, serverless functions can be used to write entries into an audit table whenever a record gets updated in the database. AWS Lambda is used alongside AWS DynamoDB[40] to create database triggers [11].

- *Serverless computing at edge.* Edge computing is described as a key driver for the serverless computing trend [55, 67]. IoT devices generate a large volume of data which need to be processed in real-time. Serverless functions at the edge can react to events, such as changes in temperature and water levels, without having to send all the data to the cloud. Amazon provides AWS Greengrass to run lambda functions at the edge on local connected devices. These functions can also run when the devices are not connected to the Internet. Deploying the functions on the edge can improve the utilization of resource-constrained devices at the edge. Furthermore, by abstracting away the infrastructure details from the developers, code can be deployed on multiple devices. Various Content Delivery Network providers, such as AWS (Lambda@Edge[41]) and Cloudflare (Cloudflare Workers[42]) provide serverless computing at their edge infrastructure. Developers can use this to perform tasks such as to modify headers, to carry out A/B testing, to inspect authorization tokens and so on.

[37] http://kubeless.io/
[38] https://www.openfaas.com/
[39] http://fission.io/
[40] https://aws.amazon.com/dynamodb/
[41] https://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html
[42] https://www.cloudflare.com/products/cloudflare-workers/

- *Media processing.* In media processing, an input file goes through various processing stages before it is ready to be served to the end user. For instance, when a raw image is uploaded by a user, a thumbnail of the same is to be generated which is then copied to a blob storage, followed by updating a database. The image might also be further processed for image recognition and other meta data extraction. All these steps are small processes but they need to run asynchronously and in parallel. Hence, serverless computing is ideal for this use case as the functions need to run only when an image is uploaded.

### 2.5.4 Benefits and Challenges

This section summarizes the benefits and challenges of serverless computing. First, we describe the benefits.

- The serverless computing platform abstracts away the server and its management from software developers. It also manages the scaling of functions on-demand [55].
- With the operating procedures abstracted away, developers can focus on writing code. Thus, they can deliver features at a faster rate and iterate faster. Moreover, development teams can be smaller as they need not have individuals working on the infrastructure. Moreover, with developers not having to manage infrastructure and scalability, the operational cost reduces significantly.
- In the case of bare metal servers and virtual servers, the practice is to reserve computing capacity, network bandwidth and storage. Developers would then deploy their applications on top of this infrastructure and were billed for the duration the infrastructure is reserved. The serverless computing model gets rid of computing capacity reservation. Developers only have to pay for the duration their code runs. Developers need not pay anything when the functions are idle. This brings a major cost benefit for developers [37].
- As discussed in Section 2.4, a monolithic application is broken down to smaller services known as microservices. This architecture pattern is very similar to serverless computing wherein an application is broken down to functions. Hence, serverless computing is complementary to the microservice architecture and shares many of its advantages.

While the advantages are many, serverless computing is still in a nascent stage. Some of the drawbacks are discussed next.

- *Cold start* is one of the major drawbacks of a serverless platform. Since functions are not always running, there can be increased latency in serving the first request as the container has to come online. Moreover, the container may also have to first install application dependencies. To avoid this effect, the serverless platform might keep the function running for some time so that it can handle subsequent requests and finally shut it down if there is no demand. Developers get around this by constantly invoking the function with a dummy execution path every few seconds to keep the function warm [20].

- When using a public cloud's serverless platform, the functions are expected to be written in certain way. This makes it difficult to switch service providers easily, resulting in vendor lock-in [47]. Moreover, because of the statelessness and event-driven feature of the functions, the functions need to use other services (such as queuing and database) provided by the cloud service provider.

- Serverless computing is in a nascent stage and it lacks standardization and maturity [5].

- Development and Operations (DevOps) practices in serverless computing have not evolved yet. Since each function is small, an application might result in hundreds (if not thousands) of functions. Each of the functions are versioned separately and hence, need to have their own deployment pipeline. To add to the complexity, all functions need not use the same runtime. This makes it difficult for the operations team to operate and monitor the services.

# Chapter 3

# Open source serverless frameworks

This chapter presents a feature evaluation of open source frameworks for implementing serverless computing. The goal of this chapter is to identify promising frameworks for our prototype implementation. Section 3.1 outlines the criteria used for comparing the frameworks. Section 3.2 discusses popular open source frameworks based on the criteria defined in Section 3.1. Finally, Section 3.3 provides some concluding remarks on selecting a framework.

## 3.1 Evaluation criteria

The features identified for comparing the serverless frameworks are described next.

- *Open source license.* The selected framework should be an open source implementation so that developers can customize features. Specifically, the license should allow to freely access, use, modify and distribute (both in modified and unmodified form) code to others.

- *Strong developer community.* The selected framework should have a strong and thriving developer community. This is evaluated by analyzing the code commit frequency, pull requests/merge requests frequency, reputation, availability of support platforms (such as web-based forums, mailing lists, Slack channel) and commercial support.

- *Programming language support.* The selected solution should have support for multiple languages. There should be out of the box support for popular languages, such as Go, Python and Node.js [30]. Additionally, it should be possible to add support for more languages. This would enable developers to also upgrade the version of a language if required.

- *Auto scaling.* Serverless functions are expected to serve infrequent and sporadic demands [5]. Thus, the framework must support scaling in order to efficiently utilize the underlying hardware even with varying incoming traffic. We also consider whether the framework supports multiple or configurable scaling criteria, such as requests per second/-queries per second (QPS), CPU and message queue size.

- *Support for multiple orchestrators.* Support for container orchestrators apart from Kubernetes (such as Docker Swarm, Nomad, etc.) provides more flexibility for both the development and operations team.

- *Function triggers.* The selected solution should support both HTTP (synchronous) and event-based (asynchronous triggers).

- *Availability of monitoring tools.* It is important that the framework has an integrated monitoring tool which can help the operations team to monitor the performance metrics of a deployed function, such as the number of invocations and execution time.

- *CLI interface.* Availability of a command line interface will greatly ease the management of functions. This will also allow for better integration with third party tools for actions, such as event-driven triggers.

- *Ease of deployment.* It should be easy to deploy the selected framework with minimal configuration changes. The steps and complexity of deployment helps evaluate how quickly a new Kubernetes cluster with the framework can be set up. This can be evaluated by following the *"Getting Started"* guide of each framework.

## 3.2 Frameworks

We choose four frameworks for evaluation: Kubeless, OpenWhisk, Fission and OpenFaaS. The frameworks were selected based on the number of GitHub stars, a mark of appreciation from users. We selected the solutions which had at least 3000 stars. This section briefly describes each framework and evaluates them based on the criteria discussed in Section 3.1.

### 3.2.1 Kubeless

Kubeless [24] is a Kubernetes-native serverless framework. It is an open source project from Bitnami[1] and is available under the Apache 2.0 license.

[1] https://bitnami.com

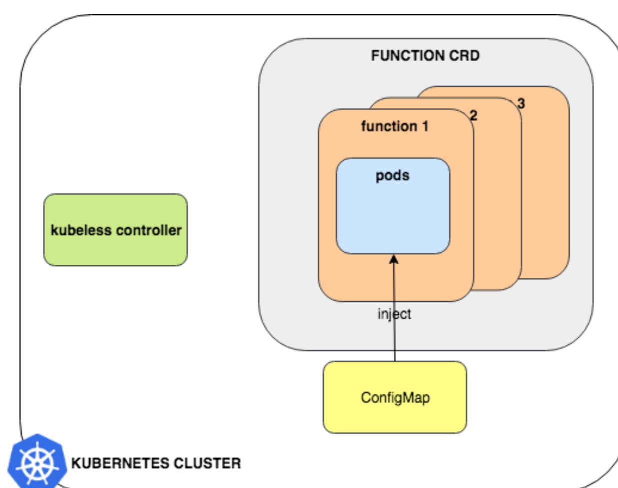



Figure 3.1: High-level architecture of Kubeless.

As shown in Figure 3.1, Kubeless uses Custom Resource Definitions (CRD) [10] to extend the Kubernetes API and create a *function* custom object in the Kubernetes API. This allows developers to use native Kubernetes APIs to interact with the functions as if they are native Kubernetes objects. Function runtimes are made available by using Deployment/Pods. Config maps are used to inject a function's code in the runtime. Services expose functions outside a pod. Function dependencies are installed by using an init container. The Kubeless controller continuously watches for any change to function objects and takes necessary action to maintain its desired state.

Currently, Kubeless supports Python (v2.7.x), NodeJS (v6.x and v8.x) and Ruby(v2.4.x). Furthermore, Kubeless allows multiple function triggers: HTTP, publish-subscribe (event-based or asynchronous) and scheduled. Event-based invocations are achieved through the use of Apache Kafka[2] and Zookeeper[3]. Kubeless uses Prometheus[4] for monitoring functions. The language runtimes are designed to generate metrics for each of the function. Autoscaling in Kubeless is achieved by using the Kubernetes Horizontal Pod Autoscaler [25] which can scale functions based on CPU usage or other custom metrics. These custom metrics are monitored by Prometheus and sent to the autoscaler. Kubeless provides both a UI and CLI interface to interact with functions. It is simple to deploy and can be setup in a matter of minutes by using the official documentation [25]. In terms of community

[2]https://kafka.apache.org
[3]http://zookeeper.apache.org
[4]https://prometheus.io

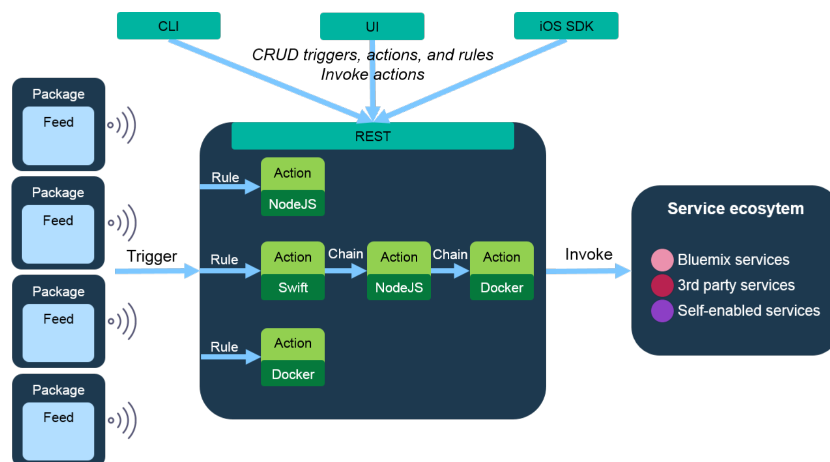



Figure 3.2: OpenWhisk: high level architecture [3].

support, Kubeless has an active open source community. The community is active on GitHub, Slack[5] and Twitter (@kubeless_sh).

### 3.2.2 Apache OpenWhisk

Apache OpenWhisk [2] is an open source, serverless computing framework initially developed by IBM and later submitted to Apache Incubator. It is available under the Apache 2.0 license. Apache OpenWhisk is also the underlying technology behind the Functions as a Service[6] product on IBM's public cloud, Bluemix[7]. Figure 3.2 illustrates the high level architecture of OpenWhisk. The OpenWhisk programming model is based on three primitives: action, trigger and rule [41]. *Actions* are stateless functions that execute code. *Triggers* are a class of events that can originate from different sources. Finally, *rules* associate a trigger with an action.

Figure 3.3 shows the system architecture of OpenWhisk. OpenWhisk relies on the following components: Nginx, Controller, CouchDB[8], Kafka, Invoker and Consul[9]. Nginx acts as a proxy for forwarding requests to the controller. The controller is a REST API gateway for all the actions a user carries out, including CRUD actions on OpenWhisk entities. A CouchDB instance is used for authentication. Once the requester is authenticated,

[5]https://kubernetes.slack.com
[6]https://www.ibm.com/cloud/functions
[7]https://www.ibm.com/cloud/
[8]http://couchdb.apache.org/
[9]https://www.consul.io/

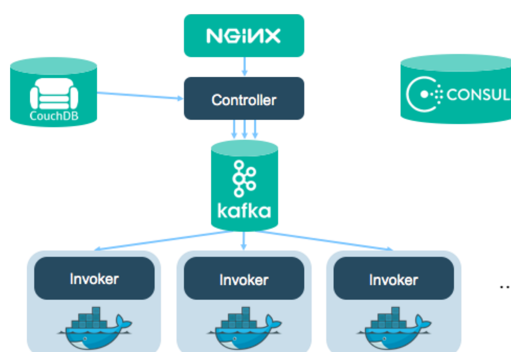



Figure 3.3: OpenWhisk system architecture [3].

the controller loads the action saved in CouchDB and passes it to the invoker. The action record mainly contains the action code, default parameters merged with the one passed and restrictions (if any). The controller and invoker communicate through Kafka. The invoker picks the action and spawns a new Docker container and injects the function code. The response is returned to the user and is also saved in the CouchDB. Consul is a distributed key-value store, which is used to manage the state of the OpenWhisk installation. The invoker decides whether to reuse an existing container (hot), to start a paused container (warm) or to launch a new container (cold) based on the state of containers in Consul.

OpenWhisk supports functions written in JavaScript, Swift, Python, PHP, Java and any binary executable. Additionally, it allows to run any custom code packaged inside a Docker container. It supports synchronous, asynchronous and schedule-based triggers for functions. Each of the components in the architecture (discussed previously) runs as a Docker container and can be scaled in real-time. OpenWhisk can be setup to run by using any container orchestrator, such as Kubernetes or Docker Swarm. However, the scalability of functions is directly managed by the OpenWhisk controller and it does not rely on native Kubernetes support. Monitoring of functions can be done by integrating statsd[10]. OpenWhisk provides CLI, REST API endpoints and iOS sdk to interact with OpenWhisk cluster. In terms of community support, OpenWhisk has an active community on GitHub[11], Slack[12], Twitter (@openwhisk) and mailing lists[13].

[10] https://github.com/etsy/statsd
[11] https://github.com/apache/incubator-openwhisk
[12] https://openwhisk-team.slack.com
[13] http://openwhisk.incubator.apache.org/contact.html

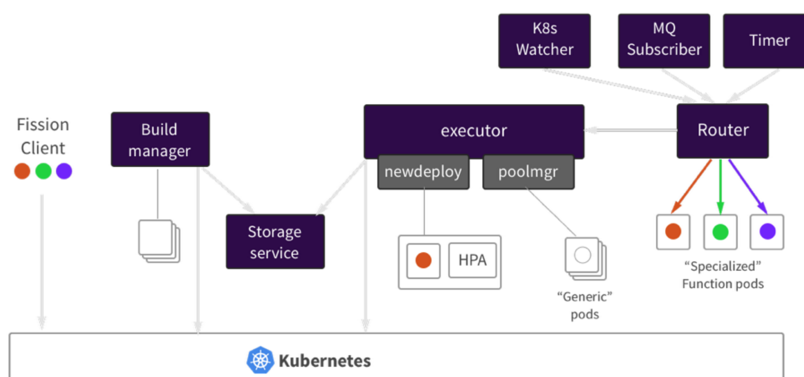



Figure 3.4: Fission system architecture [14].

### 3.2.3 Fission

Fission [13] is an open source, serverless computing framework for Kubernetes from Platform9[14] and is available under the Apache 2.0 license. Fission is built specifically for Kubernetes and uses many Kubernetes native concepts. Figure 3.4 illustrates the system architecture of Fission. The key components in Fission are: controller, environment, executor, router, kubewatcher, and logger. The controller contains the CRUD APIs for functions, HTTP triggers, Kubernetes event watches and environments. Environment containers are language-specific containers. Each environment container must contain an HTTP server and a loader for functions. The executor is responsible for creating function pods. There are two types of executor: *poolmgr* and *newdeploy*. Poolmgr creates a set of generic pods as soon as an environment is created. These are called warm containers. When a function is invoked, one of the pods is taken from the environment and used for execution. After a certain period of inactivity, the pod is cleaned up and returned to the pool. A poolmgr is ideal for low latency but cannot be auto scaled. On the other hand, newdeploy supports auto scaling. It creates a Kubernetes deployment along with a service and a Horizontal Pod Autoscaler for function execution. The router is responsible for forwarding HTTP requests to function pods. Kubewatcher watches the Kubernetes API and invokes functions linked with the watches. The logger is responsible for forwarding function logs to a centralized location. Currently, Fission stores logging on

[14]https://platform9.com/

InfluxDB[15]. For monitoring, Fission recommends Istio[16] which can be integrated with Prometheus and Grafana[17]. Fission has both a web-based UI and a command line interface for observing and managing Fission [15].

Fission allows developers to write functions in Python, NodeJS, Ruby, Perl, Go, Bash, C# and PHP. It also allows to run custom containers. Fission supports three types of triggers: HTTP trigger, time-based trigger and Message Queue (MQ) trigger. An HTTP trigger can invoke functions through incoming HTTP requests, including GET, POST, PUT, DELETE and HEAD. Time-based triggers are similar to cron jobs and are triggered periodically. MQ triggers are for asynchronous function calls. Here functions listen to a topic and invoke a function whenever a message arrives. Currently, nats-streaming[18] and azure-storage-queue[19] are supported message queues. Fission also provides a workflow-based serverless composition framework called Fission Workflows which creates a flowchart of functions allowing multiple functions to be composed together. This is in active development now and not stable yet [16]. Fission uses the Horizontal Pod Autoscaler (HPA) to scale function based on CPU utilization. Custom metrics are not supported at the moment. In terms of community support, Fission has an active open source community. The community is active on GitHub[20], Slack[21] and Twitter (@fissionio).

### 3.2.4 OpenFaaS

OpenFaaS [28] is an open source, serverless computing framework for Kubernetes and is available under the MIT license. It recently received the backing of VMWare [33]. OpenFaaS is built with focus on ease of use, portability, simplicity and openness. As shown in Figure 3.5, OpenFaaS is composed of two key components: an API Gateway and a function watchdog. Prometheus is used for recording metrics. This framework supports multiple container orchestrators, including Kubernetes, Docker Swarm and Nomad. It can be easily extended to support other Docker orchestrators. Figure 3.6 illustrates how the components of OpenFaaS interact with each other. The API gateway is a RESTful microservice that provides an external route for a function. The

[15]https://www.influxdata.com/

[16]https://github.com/istio/istio

[17]https://github.com/grafana/grafana

[18]https://nats.io/documentation/streaming/nats-streaming-intro/

[19]https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-azure-and-service-bus-queues-compared-contrasted

[20]https://github.com/fission/fission

[21]https://fissionio.slack.com

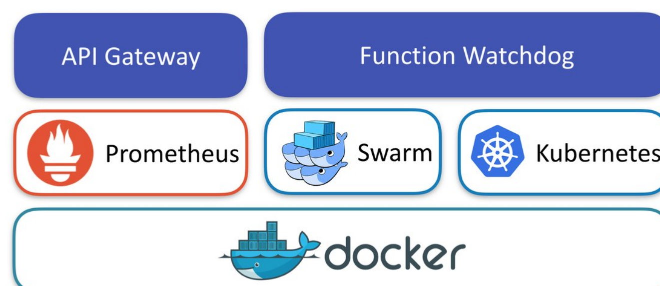



Figure 3.5: OpenFaaS system architecture [29].

API gateway relies on native functionalities provided by the chosen Docker orchestrator. For this, the API gateway connects with respective plugin for the chosen orchestrator, for example, faas-netes[22] for Kubernetes. The gateway also records various function metrics in Prometheus. It is also responsible for scaling functions based on alerts received from Prometheus through an AlertManager. The function watchdog is packaged together with the function and is the entry point for every function call.

OpenFaaS allows developers to write functions in any programming language of their choice. The functions have to be packaged in a Docker image. For ease of use, OpenFaaS has templates for C#, Go, NodeJS, Python and Ruby. However, developers can create their own templates. Two types of function triggers are supported: HTTP and event-based. The event-based trigger uses nats-streaming. Development is underway for integration with Kafka. In terms of community support, OpenFaaS has an active open source community. The community is active on GitHub, Slack[23] and Twitter (@openfaas). OpenFaaS has a distributed architecture and each component has its own repository, which can be accessed from the main GitHub repository[24].

[22]https://github.com/openfaas/faas-netes/
[23]https://openfaas.slack.com
[24]https://github.com/openfaas

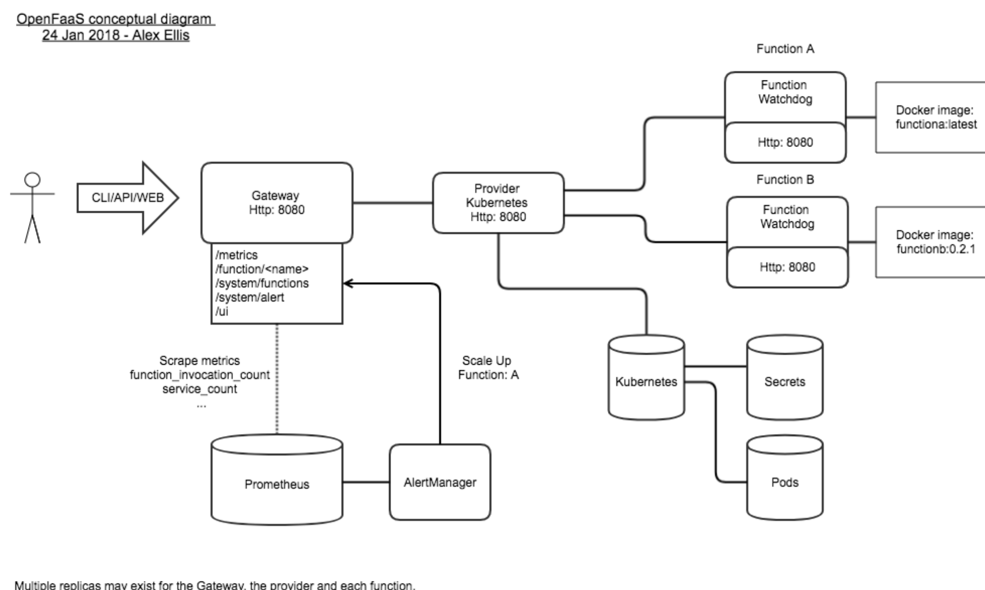



Figure 3.6: OpenFaaS: interaction between components [17].

## 3.3 Summary and comparison

In this section, we summarize the discussion of the open source serverless frameworks based on the criteria described in Section 3.1. Table 3.1 presents a summary of the features of the frameworks. All four solutions are open source under either the Apache 2.0 or MIT license. Given the number of GitHub stars and forks, OpenFaaS has the largest developer community. The other three solutions also have a relatively solid developer community base. The considered frameworks support the popular languages as well as custom containers. However, OpenFaaS architecture is not designed around runtimes. All the functions are packaged by using templates[25] which are completely customizable [8]. Templates contains function templates which are used to initialize function code. They also contain a Dockerfile which is used to build and package functions as a Docker image. Templates are just wireframes around which OpenFaaS functions are built and can be completely customized. This approach makes it easy for developers to design their own templates with any language and dependencies. The templates can then be shared across multiple teams. As for scaling of functions, Fission cur-

[25]https://github.com/openfaas/templates

rently supports auto scaling based on CPU utilization, whereas OpenWhisk supports scaling based on number of requests. On the other hand, Kubeless and OpenFaaS allow auto scaling based on custom metrics. Kubeless and Fission are completely Kubernetes-native solutions and do not support any other container orchestrator. On the other hand, OpenWhisk does not rely on native features of any orchestrator. OpenFaaS supports multiple container orchestrators (Kubernetes, Docker Swarm, Nomad) and uses the native functionalities of the chosen orchestrator. Kubeless, OpenWhisk and Fission support HTTP, event and schedule-based function triggers. OpenFaaS currently supports HTTP and event-based triggers; however, support for schedule-based triggers is in the pipeline. Deploying OpenFaaS, Fission and Kubeless frameworks on Kubernetes v1.9.6 (latest version available at the time of this thesis work) was easy. On the other hand, deploying OpenWhisk on the same Kubernetes version was cumbersome. We tried deploying OpenWhisk by following the *"Installation Guide"* guide but encountered errors a few minutes into the installation process [22]. Kubeless and OpenFaaS have native Prometheus integration. This gives access to a rich set of monitoring parameters. On the other hand, OpenWhisk and Fission have recommended integration with statsd [23] and Istio [12] respectively. All the frameworks provide a CLI to interact with functions.

In conclusion, Kubeless has the simplest architecture and makes use of Kubernetes features natively as much as possible. OpenFaaS is highly extendable and supports mutiple container orchestrators, thus providing a lot of flexibility.

| Feature | Kubeless | OpenWhisk | Fission | OpenFaaS |
|---|---|---|---|---|
| **Open source** | Yes | Yes | Yes | Yes |
| **License** | Apache 2.0 | Apache 2.0 | Apache 2.0 | MIT |
| **Programming languages** | Python, NodeJS, Ruby, PHP and custom runtimes | JS, Swift, Python, PHP, Linux binary and custom containers | Python, NodeJS, Ruby, Perl, Go, Bash, C#, PHP and custom containers | Any language that can be executed from bash |
| **Auto scaling** | Yes | Yes | Yes | Yes |
| **Auto scaling metric** | CPU, QPS and custom metrics | QPS, managed by OpenWhisk | CPU utilization | CPU, QPS and custom metrics |
| **Container orchestrator** | Kubernetes native | No orchestrator required, Kubernetes supported | Kubernetes native | Kubernetes, Docker Swarm, Nomad, extendable to other orchestrators |
| **Function triggers** | HTTP, event, schedule | HTTP, event, schedule | HTTP, event, Kubernetes watch, schedule | HTTP, event |
| **Message queue integration** | Kafka | Kafka | NATS, Azure storage queue | NATS, Kafka |
| **Recommended monitoring tool** | Prometheus | statsd | Istio | Prometheus |
| **CLI Support** | Yes | Yes | Yes | Yes |
| **Industry support** | Bitnami | IBM, Adobe, RedHat and Apache Software Foundation | Platform9 | VMWare |
| **GitHub stars** | 3036 | 3312 | 3425 | 10674 |
| **GitHub forks** | 278 | 630 | 278 | 766 |
| **GitHub contributors** | 62 | 120 | 65 | 68 |

Table 3.1: Overview of features supported by serverless frameworks.

# Chapter 4

# Evaluation

The goal of this chapter is to evaluate the performance of the serverless computing frameworks presented in the previous chapter. Section 4.1 describes the methodology used to evaluate the frameworks. Section 4.2 describes the development and deployment experience on Fission, Kubeless and OpenFaaS. Section 4.3 evaluates the different modes of invoking function in OpenFaaS. Section 4.4 compares the performance of the three different serverless frameworks.

## 4.1 Methodology

This section outlines the experiments used to evaluate the serverless frameworks, Kubeless, Fission and OpenFaaS. We no longer consider OpenWhisk due to issues faced in setup and its almost minimal dependence on Kubernetes for container orchestration. Furthermore, a performance evaluation of OpenWhisk is presented in [74]. We carry out the following experiments.

- First, we evaluate the developer experience; in particular, we outline how to develop, deploy and manage functions on Fission, Kubeless and OpenFaaS. To this end, we design a simple function and then deploy the function to a Kubernetes cluster for each considered framework.
- OpenFaaS uses three different (watchdog) modes for invoking a function: HTTP, serializing and streaming. First, we evaluate the performance of these modes to find the best performing mode. Specifically, we measure the response time for a client request in each mode under different workloads. To this end, we write a simple function and deploy the function by using each watchdog mode. The function simply takes a string as input and prints the same as the response. This function was

selected so as not to add any performance overhead owing to function logic. The code for the functions used in the experiments are listed in Appendix D. We use Apache Benchmark[1] to send concurrent requests to the functions. In each experiment run, we send 50,000 requests with either 1, 2, 5 or 10 concurrent users. We also describe the monitoring of functions and reading of logs with each mode. Monitoring and logging are important aspects in ensuring the stability of a system. Developers need to constantly monitor the state of a system through various metrics to ensure that the system is healthy. They also need access to application and system logs for debugging and auditing.

- Finally, we evaluate the performance of function invocation in OpenFaaS, Fission and Kubeless. Specifically, we measure the response time and the ratio of successful responses (those with HTTP 2xx response code) under different workloads. To this end, we write a simple function that returns the input string (similar to the previous experiment) and deploy the function on each framework. Again, we represent different workloads by using Apache Benchmark to send 10,000 requests with either 1, 5, 20, 50 or 100 concurrent users. We disable auto scaling of the functions in order to analyze the performance of each framework architecture without the impact of scaling, i.e., the possible increase in response time when creating a new container or pod. Thus, a fixed number of function replicas (1, 2, 5 or 10) is used in each experiment. Since this experiment requires more computing resources than available in the previous experimental setup, we run these experiments on Google Kubernetes Engine[2] (GKE). The deployment on GKE is similar to running the framework on a custom Kubernetes cluster.

## 4.2 Ease of development

As discussed in Section 2.5, one of the important benefits of serverless computing is that it allows developers to focus on writing code and delivering features at a faster rate. Hence, in this section, we describe the steps needed to develop and deploy a new function in each framework. We developed simple functions in Go programming language and deployed it on each framework.

*Fission* provides a CLI tool `fission` to interact with the Fission APIs. This makes it easy and convenient to build and deploy functions. First, we

[1] https://httpd.apache.org/docs/2.4/programs/ab.html
[2] https://cloud.google.com/kubernetes-engine/

create a function as shown in Appendix A.1. The function has very less boilerplate code. The next step is to create an *environment* for Go. Environments contain the information related to language and runtime for our function. Next, we create the function using the environment created in the previous step. While creating an environment, we can also use the option of *builder*, which is used for building from source code [13]. The commands discussed here are listed in Appendix A.2. Fission also provides a mechanism to create a declarative specification for creating all the components [32]. This allows to manage Fission components using a yml file instead of CLI commands. This is better than scripting using the CLI tool `fission`. Furthermore, the availability of the CLI and the declarative syntax makes it easy to design and build a CI/CD (continuous integration/continuous delivery) pipeline.

*OpenFaaS* provides a CLI tool `openfaas-cli` to interact with the OpenFaaS APIs. This makes it easy and convenient to build and deploy functions to an OpenFaaS cluster. First, a function template is generated by using the `openfaas-cli` (Appendix C.1). This step generates a source file with a simple function that can be edited to add the actual business logic. The function has very less boilerplate code. This step also generates a yml file with the function configuration, which can be used to deploy and manage functions (Appendix C.1.2). Once the business logic is written, the function is deployed in the Kubernetes/OpenFaaS cluster by using the `openfaas-cli` commands: `build`, `push` and `deploy`. The exact commands are listed in Appendix C.2. These commands build Docker images, push them to the image repository (listed in the yml file) and deploy them to the Kubernetes cluster. The CLI tool also makes it easy to design and build a CI/CD pipeline. Single line commands can be used to build, push and deploy functions.

*Kubeless* also provides a CLI tool `kubeless` to interact with the Kubeless components. First, we create a function as shown in Appendix B.1. Here too, the function has very less boilerplate code. The next step is to deploy the function. This is done by running a single line command (listed in Appendix B.2). Unlike OpenFaaS and Fission, Kubeless does not have a native declarative specification. Finally, the `kubeless` CLI can be integrated with a CI/CD pipeline.

Finally, secrets are an important component of application development and management. In Kubernetes, secrets are of two types: application secrets and ImagePullSecrets[3]. Application secrets are sensitive information used by the application logic, such as database credentials, API keys, etc. ImagePullSecrets are used to authenticate to a private Docker registry to pull images. All the three frameworks provide a mechanism to handle these

[3]https://kubernetes.io/docs/concepts/configuration/secret/

| CPU | 2 cores |
|---|---|
| RAM | 2000 MB |
| Operating System | Ubuntu 16.04.04 LTS |
| Kubernetes Version | 1.10.1 |
| Docker Version | 1.11.2 |

Table 4.1: Configuration of Kubernetes nodes.

secrets by interacting with Kubernetes secrets objects. Users can refer to Kubernetes secrets while deploying functions.

All the considered frameworks simplify the software development process for developers by allowing them to quickly develop and deploy functions to the production environment.

## 4.3 Comparison of Watchdog modes in OpenFaaS

Next, we compare the response time of functions with the three different watchdog modes in OpenFaaS.

### 4.3.1 Setup

The complete system architecture of the prototype system is outlined in Figure 4.1. As shown in the figure, the system comprises of 3 virtual machines. One of the VMs is the Kubernetes master node and the other two are the worker nodes. The virtual machines (nodes) are setup by using the VirtualBox software[4]. Table 4.1 illustrates the configuration of each of the nodes. The host machine has a 2.3 GHz 4-core CPU, 16 GB RAM and runs macOS 10.13.5. Calico v2.0[5] is used for pod networking in the Kubernetes cluster.

#### Kubernetes installation

We use Kubeadm[6] for creating the Kubernetes cluster. The Kubeadm toolkit provides an easy and secure way to bootstrap a Kubernetes cluster. The `kubeadm init` command is used to quickly bootstrap the master node. This creates the necessary components (pods) required for the master and generates the token required by the worker nodes to connect to cluster. This

[4]https://www.virtualbox.org/
[5]https://docs.projectcalico.org/v2.0/getting-started/kubernetes
[6]https://kubernetes.io/docs/tasks/tools/install-kubeadm/

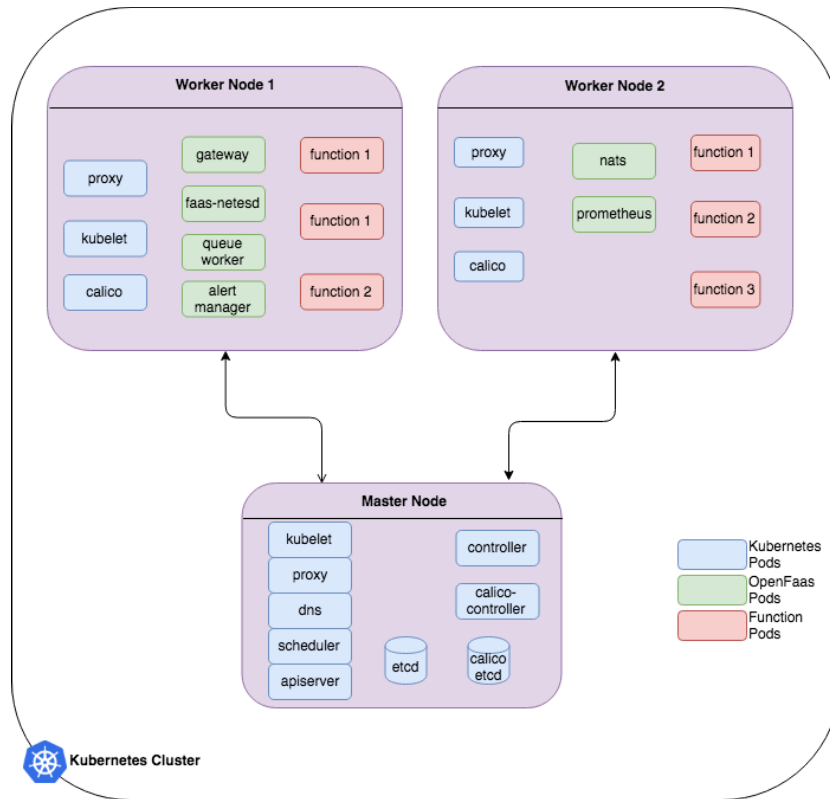



Figure 4.1: System architecture of OpenFaaS on a Kubernetes cluster.

process also generates the necessary configuration file (*/etc/kubernetes/admin.conf*) required to connect to the API server. Then Calico[7] is installed on the cluster. Calico is required to provide pod networking within the Kubernetes cluster. Calico etcd and Calico controller are installed on the master node and Calico containers are installed on all the nodes. Once this is done, individual worker nodes are configured. The worker nodes join the Kubernetes cluster by using the token generated by the master node. For our implementation, two worker nodes are configured in the cluster.

### OpenFaaS installation

Next OpenFaaS is installed on the Kubernetes cluster. The core components of OpenFaaS are the following:

- The *gateway* is the main entry point for clients to interact with the

[7]https://docs.projectcalico.org/v2.0/getting-started/kubernetes/

| Component | Version |
|---|---|
| Gateway | 0.7.9 |
| faas-netes | 0.5.1 |
| Prometheus | 2.2.0 |
| Alert Manager | 0.15.0-rc.0 |
| Nats streaming Server | 0.6.0 |
| Queue Worker | 0.4.3 |
| Faas-cli | 0.6.9 |

Table 4.2: OpenFaaS core components deployed on Kubernetes.

functions (deploy, scale, remove and invoke).

- *faas-netes* enables Kubernetes to be used as the backend.
- *Prometheus* is used for saving various metrics required for monitoring OpenFaaS.
- *AlertManager* sends alerts to Gateway to scale functions based on usage data saved in Prometheus.

As shown in Figure 4.1, all the components are deployed with a replica count of 1. The gateway and Prometheus are both exposed outside the Kubernetes cluster through NodePort[8]. For asynchronous function invocation[9], a *nats-streaming* server and *queue worker* are deployed. All the components are installed by using the Kubernetes YAML file provided at faas-netes GitHub project[10]. Additionally, the *faas-cli* tool was installed for accessing OpenFaaS APIs. This allows functions to be created and deployed by using the OpenFaaS CLI. The Gitlab CI pipeline is used for managing the CI/CD (continuous integration and deployment) pipeline. All OpenFaaS functions are deployed in *openfaas-fn* namespace. Table 4.2 details the version of each component in our setup.

Functions in OpenFaaS run inside a container (or pod in Kubernetes). Along with the function code or binary, a function container contains an OpenFaaS component called a *watchdog*. The watchdog is the entry point for all function calls. It invokes the actual function, parses the response received from the function and finally sends it back to the client.

[8] https://kubernetes.io/docs/concepts/services-networking/service/

[9] Event driven programming is at the heart of serverless computing and this can be achieved by asynchronous function invocation.

[10] https://github.com/openfaas/faas-netes/tree/master/yaml

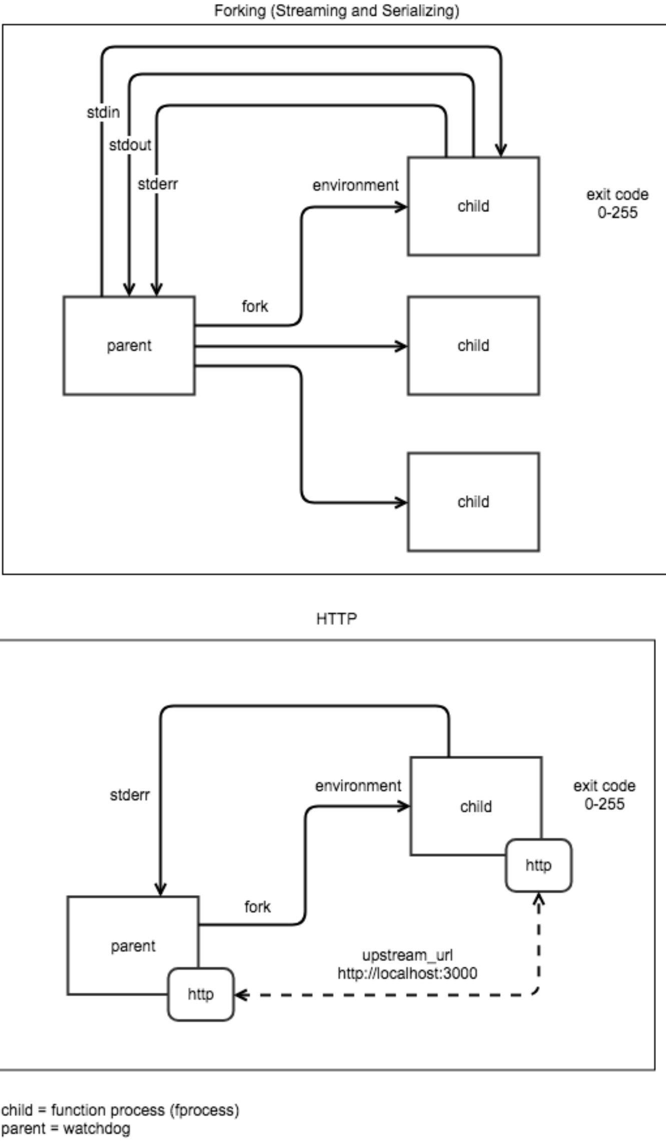



Figure 4.2: OpenFaaS watchdog modes [34].

The watchdog can be configured with four different modes: serializing, HTTP, streaming and Afterburn[11]. These modes affect how the functions are invoked. In both the *serializing* (classic) and *streaming* mode, the watchdog forks one process per incoming request. This process can be a function binary or a language runtime (for instance, Python or Node.js). In the *serializing* mode, the watchdog reads the entire request into memory and then passes it to the function. It then reads the stdout and serializes this as a response to the client. On the other hand, in the streaming mode, the function input is directly linked to stdin and the stdout to the response. The function reads the stdin and writes the response to stdout. The watchdog then makes the response available to the client as soon as it is available in stdout. In the *HTTP* mode, a process is forked as soon as the function container comes up and the watchdog starts. Function requests are sent to a specific port being listened to by the function. As shown in Figure 4.2, both the function and watchdog run as servers and interact over HTTP. The main difference between *HTTP* mode and previous two modes is that *HTTP* mode does not fork a process per request.

In our experiments, we use different watchdog binaries for different modes. The original watchdog binary does not support streaming and HTTP modes. For these two modes, we use the new watchdog binary[12] and for serializing we use the original watchdog binary[13]. Hence, we also refer to the serializing mode as classic mode. Figure 4.2 illustrates the different modes.

### 4.3.2 Results

We evaluated three different watchdog modes of OpenFaaS: serializing (or classic), HTTP and streaming. Our goal is to find the best performing watchdog mode. Figure 4.3 shows the average response time of the OpenFaaS function with these modes. The figure reports the average values obtained from ten runs as well as the corresponding standard deviations, shown as whiskers in the plots. We observe that response time is lowest for the HTTP mode. This is due to a process being forked for each incoming request in the classic and streaming mode. Forking a process is a resource intensive task. On the other hand, in HTTP mode a new function fork is created only when the function container is launched. Subsequent requests are served through the same forked process and the communication between watchdog and the function is done over HTTP.

[11]Afterburn is being deprecated and hence we do not discuss it further

[12]https://github.com/openfaas-incubator/of-watchdog

[13]https://github.com/openfaas/faas/tree/master/watchdog

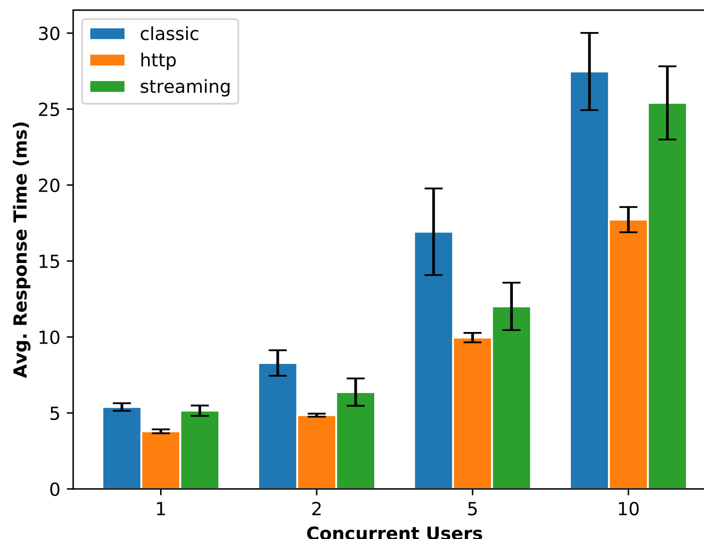


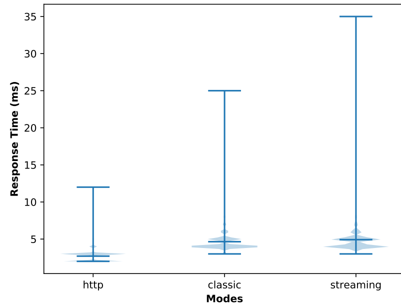

Figure 4.3: Average response time (in ms) for different OpenFaaS watchdog modes.

Figure 4.4 shows the distribution of response times for 1, 2, 5 and 10 concurrent users across different watchdog modes. The results are from a single experimental run of 50,000 requests made for each mode. We remove the top 15 response time values (0.03%) from each mode. We consider the top 0.03% of response times as outliers. The reason for the outliers can be attributed to sufficient replicas not being available to process the requests. The plots show that the response times are mostly around the median and within a narrow range of values. This is because OpenFaaS automatically scales functions as the volume of requests per second increases. This shows that the scalability provided by OpenFaaS platform along with Kubernetes is able to handle the volume of requests and ensure that response latency does not increase unreasonably.

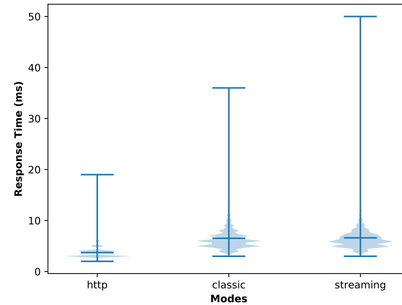
OpenFaaS is tightly integrated with Prometheus. Prometheus is used to record various metrics, such as response time, number of function replicas, number of function invocations and response codes. Prometheus can be further integrated with dashboard tools, such as Grafana[14] to give a rich dashboard of the OpenFaaS functions.

By default OpenFaaS logs are not written to container logs. This can

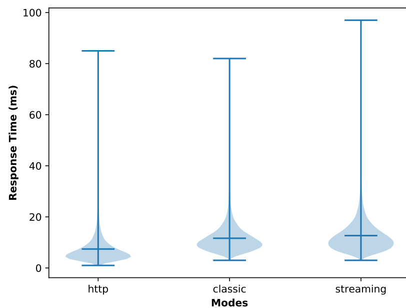
[14]https://grafana.com



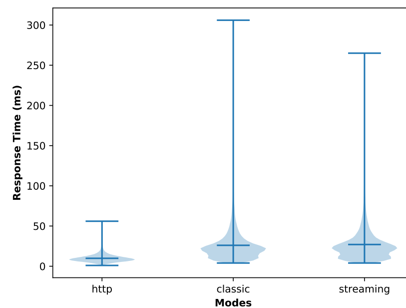

(a) 1 concurrent user




(b) 2 concurrent users




(c) 5 concurrent users




(d) 10 concurrent users

Figure 4.4: Density of response time for different watchdog modes in OpenFaaS.

be enabled by setting *write_debug* environmental variable to true. Logs can also be checked by using the *kubectl logs* command of Kubernetes. The logs discussed so far are system level logs. Additionally, developers would like to write and analyze application level logs. Here the choice of the watchdog mode is important. In the classic mode, anything written to stderr by the function is returned back to the client. On the other hand, in the other two modes, application logs are recorded and written to the container logs.

## 4.4 Performance of the different frameworks

In this section, we evaluate the performance of functions deployed on Kubeless, Fission and OpenFaaS.

### 4.4.1 Setup

We evaluate the performance of the three serverless frameworks with concurrency as high as 100 and the function replica count as high as 50. The function replica counts are kept fixed and no scaling is used. The setup used in the previous experiment was not sufficient for this experiment. Thus, we use GKE with the necessary computing resources.

#### GKE Setup

In GKE, we use Kubernetes version 1.10.4-gke.2 (latest version available at the time of the experiment). Three worker nodes are used in the cluster, each node having 2 vCPUs and 7.5 GB memory. We use Container-Optimized OS as the operating system for the worker nodes. This operating system is optimized for running containers and Kubernetes on Google Cloud Platform [6]. The cluster is setup in the *europe-north1*[15] region and all the nodes are located in zone *a*. We also set up a VM instance to invoke the functions deployed on the GKE cluster. The instance is setup in the same zone as the GKE cluster to minimize network latency. The instance has 2 vCPUs, 7.5 GB memory and ran Debian GNU/Linux 9.4 (stretch) OS.

#### OpenFaaS installation

We follow the same steps as described in Section 4.3.1 for deploying OpenFaaS. However, we use only the HTTP watchdog mode. This is motivated by the results in Section 4.3.2 which indicate that HTTP mode performs better than the other modes.

#### Kubeless installation

We follow the steps outlined in the official Kubeless *Quick Start*[16] for installing Kubeless on GKE. We deploy Kubeless v1.0.0-alpha.6 with Role Based Access Control (RBAC) enabled. First, we create the kubeless namespace and install the kubeless controller. Following this, we install components required for exposing Kubeless function outside the GKE cluster. Kubeless uses Kubernetes Ingress[17] to provide routing for the functions. We use the Nginx Ingress controller by following the instructions in the *Installation Guide*[18]. Next the functions are deployed and they are scaled to a fixed

[15] https://cloud.google.com/compute/docs/regions-zones
[16] https://kubeless.io/docs/quick-start/
[17] https://kubernetes.io/docs/concepts/services-networking/ingress/
[18] https://kubernetes.io/docs/concepts/services-networking/ingress/

replica size. Finally, *triggers*[19] are created for the functions to expose them outside the GKE cluster.

**Fission installation**

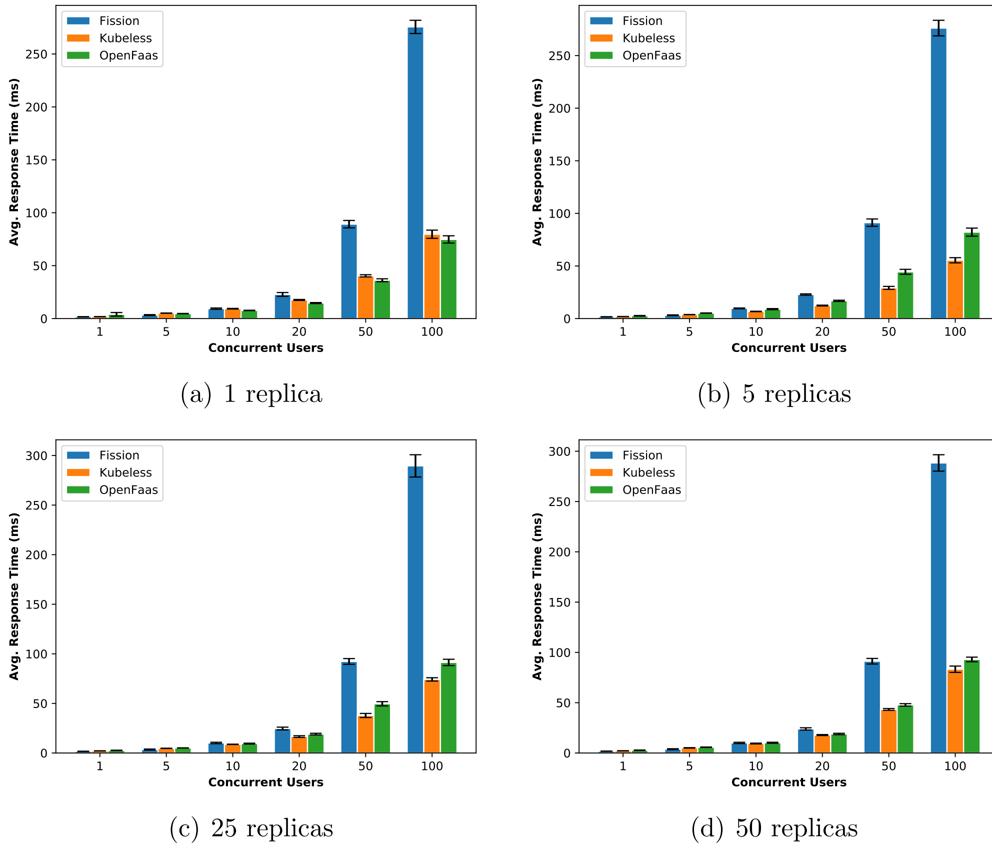
We follow the steps outlined in the official Fission *Installation Guide*[20] for installing Fission on GKE. We then create an *environment* for Go, *package*, route and the actual function. Environments contains the information related to language and runtime for our function. The package provides a mechanism to build the function from source code [13]. *Route*s are created to invoke Fission functions through HTTP. Finally, the functions are created by defining the appropriate minscale and maxscale. Though we can use either *pool based executor* and *newdeploy*, we chose the *newdeploy* executor [7] as this supports autoscaling of functions.

### 4.4.2 Results

Figure 4.5 shows the average response time of the functions in the three considered frameworks. The figure reports the average values obtained from ten iterations as well as the corresponding standard deviations, shown as whiskers in the plots. We observe that the response time is lowest for Kubeless and highest for Fission. The difference in the response time increases with more load, i.e., as the concurrency of requests increases. We believe the lower performance of Fission is due to the *router* component through which all HTTP requests are routed. On the other hand, Kubeless relies on native Kubernetes components wherever possible. For instance, Kubeless uses Kubernetes ingress for exposing function outside the Kubernetes cluster. Our experiments indicate that the performance using an ingress is better than the other approaches. The response time for OpenFaaS is larger than that of Kubeless. However, they are very close to each other. Moreover, OpenFaaS is not affected by the performance degradation seen in Fission at higher loads, specifically for 50 and 100 concurrent users.

We also monitor the response codes returned by the function invocations. Since the functions are invoked by using HTTP triggers, we track all responses with a response code of 2xx as successful. This information is obtained from the Apache benchmark tool. Table 4.3 presents the success ratio of all function invocations for the scenarios described above. We note that Kubeless has a 100% success ratio across all scenarios (i.e., all HTTP responses were successfully received). The success rate for OpenFaaS is less

[19] https://kubeless.io/docs/http-triggers/
[20] https://docs.fission.io/0.8.0/installation/installation/




(a) 1 replica

(b) 5 replicas

(c) 25 replicas

(d) 50 replicas

Figure 4.5: Average response time (in milliseconds) for different serverless frameworks with 1, 5, 25 and 50 function replicas.

than 97% for 100 concurrent users for 5, 25 and 50 function replicas. We also note that for OpenFaaS with 50 and 100 concurrent users, the success ratio with 1 function replica is higher than with 5, 25 or 50 function replicas. This may be attributed to the design of OpenFaaS. Every function call has to go through the gateway, faas-netes, watchdog before reaching the function. This introduces multiple hops and multiple points of failure. When there are more function replicas, the gateway and faas-netes may introduce bottlenecks leading to the observed increase in failed responses. Fission has a stable success ratio across all concurrent users and replica counts. Though, the success rate has decreased with the increase in concurrent users, the difference is minimal. In comparison to OpenFaaS, its performance in case of 50 and 100 concurrent users have been better. This may be because Fission has fewer hops during function invocation in comparison to OpenFaaS.

| | | **Concurrent users** | | | | | |
|---|---|---|---|---|---|---|---|
| | **# rep.** | **1** | **5** | **10** | **20** | **50** | **100** |
| **Kubeless** | **1** | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| | **5** | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| | **25** | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| | **50** | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| **Fission** | **1** | 100.00 | 99.90 | 99.84 | 99.78 | 99.54 | 99.32 |
| | **5** | 100.00 | 99.89 | 99.84 | 99.78 | 99.64 | 99.45 |
| | **25** | 100.00 | 99.89 | 99.85 | 99.77 | 99.48 | 99.19 |
| | **50** | 100.00 | 99.88 | 99.81 | 99.79 | 99.61 | 99.31 |
| **OpenFaaS** | **1** | 99.95 | 99.99 | 99.91 | 99.58 | 98.73 | 98.27 |
| | **5** | 100.00 | 99.99 | 99.93 | 99.61 | 97.66 | 96.23 |
| | **25** | 100.00 | 100.00 | 99.92 | 99.67 | 97.76 | 96.04 |
| | **50** | 100.00 | 100.00 | 99.93 | 99.61 | 97.48 | 96.52 |

Table 4.3: Success ratio (in %) of all requests for different serverless frameworks.

# Chapter 5

# Conclusion

Over the years, server-side computing has evolved from bare metal servers to virtualization and most recently to serverless computing. In this thesis, we presented a serverless computing framework based on Kubernetes. We discussed various use cases where serverless computing is ideal. Our main motivation for this work has been to set up a custom serverless computing infrastructure instead of using the solutions provided by public cloud service providers. This can help to bring the many benefits of serverless computing to the private cloud. We carried out a feature evaluation of open source serverless computing solutions: Kubeless, OpenWhisk, Fission and OpenFaaS. In terms of features, we found OpenFaaS to be the most flexible in terms of support for multiple container orchestrators and multiple languages. It also has the largest GitHub community based on the number of stars and forks.

Next, we evaluated a few serverless computing frameworks based on Kubernetes. To this end, we set up three different solutions: OpenFaaS, Kubeless and Fission. We evaluated the ease of developing and deploying functions on these frameworks. We also discussed how the cluster can be monitored through both system and application logs. We then evaluated the different modes of invoking functions in OpenFaaS. We concluded that the HTTP mode has the lowest response time and used this mode in the rest of the evaluation. Finally, we compared the performance of the three considered serverless frameworks. We found that Kubeless has the best performance among the three frameworks, both in terms of response time and ratio of successful responses.

The concepts presented in this thesis can be extended in the following ways.

- In our experiments, we have a maximum of three worker nodes and a single master node. This can be extended with more master nodes and worker nodes to provide high availability and more processing power.

An interesting avenue for future research would be a performance comparison with commercial solutions from cloud providers.

- OpenFaaS provides functionality for asynchronous function calls and event based processing. The frameworks can be extended to support Cloud Events. Cloud Events is a specification by Cloud Native Computing Foundation with an aim to standardize event description so that it can be consumed across multiple platforms [4].

- Our work can be extended to IoT gateways which is an interesting use case for serverless computing. In fact, IoT gateways have limited computing capability and need to serve bursty workloads. Serverless computing can efficiently manage this kind of workload [5]. The architecture used in this thesis can be replicated on an IoT gateway. Moreover, serverless computing can be used along with gateway as a service [65], where serverless computing can be used to deploy various functions as services on an IoT gateway. Thus, a proof of concept can be developed for a serverless computing framework on an IoT gateway.

# Bibliography

[1] About storage drivers — docker documentation. https://docs.docker.com/storage/storagedriver/#images-and-layers. (Accessed on 02/26/2018).

[2] Apache openwhisk is a serverless, open source cloud platform. https://openwhisk.apache.org/. (Accessed on 04/27/2018).

[3] apache/incubator-openwhisk: Apache openwhisk is a serverless event-based programming service and an apache incubator project. https://github.com/apache/incubator-openwhisk. (Accessed on 03/21/2018).

[4] Cloudevents. https://cloudevents.io/. (Accessed on 06/17/2018).

[5] CNCF serverless whitepaper v1.0. https://github.com/cncf/wg-serverless/tree/master/whitepaper. (Accessed on 02/18/2018).

[6] Container-optimized os — google cloud. https://cloud.google.com/container-optimized-os/. (Accessed on 07/05/2018).

[7] Controlling function execution :: Serverless functions for kubernetes. https://docs.fission.io/0.8.0/concepts/executor/. (Accessed on 07/05/2018).

[8] Create functions - openfaas. https://docs.openfaas.com/cli/templates/. (Accessed on 06/27/2018).

[9] Create functions - openfaas. https://docs.openfaas.com/cli/templates/#43-use-your-own-templates. (Accessed on 06/14/2018).

[10] Custom resources - kubernetes. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/. (Accessed on 06/27/2018).

[11] Dynamodb streams and aws lambda triggers - amazon dynamodb. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.Lambda.html. (Accessed on 06/27/2018).

[12] Enabling istio on fission :: Serverless functions for kubernetes. https://docs.fission.io/0.8.0/tutorial/enabling-istio-on-fission/. (Accessed on 06/27/2018).

[13] Fission :: Serverless functions for kubernetes. https://docs.fission.io/. (Accessed on 06/26/2018).

[14] Fission: a faas for kubernetes @ scale16x // speaker deck. https://speakerdeck.com/soamvasani/fission-a-faas-for-kubernetes-at-scale16x. (Accessed on 03/27/2018).

[15] fission/fission: Fast serverless functions for kubernetes. https://github.com/fission/fission. (Accessed on 03/26/2018).

[16] fission/fission-workflows: Workflows for fission: Fast, reliable and lightweight function composition for serverless functions. https://github.com/fission/fission-workflows. (Accessed on 03/26/2018).

[17] Gateway - openfaas. https://docs.openfaas.com/architecture/gateway/. (Accessed on 05/26/2018).

[18] How nodes works. https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/. (Accessed on 26/01/2018).

[19] How swarm works. https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/. (Accessed on 26/01/2018).

[20] How to keep your lambda functions warm ? a cloud guru. https://read.acloud.guru/how-to-keep-your-lambda-functions-warm-9d7e1aa6e2f0. (Accessed on 06/27/2018).

[21] I heard about the 8 fallacies of distributed computing few years ago. https://www.rgoarchitects.com/Files/fallacies.pdf. (Accessed on 06/27/2018).

[22] incubator-openwhisk-deploy-kube/readme.md at master · apache/incubator-openwhisk-deploy-kube. https://github.com/apache/incubator-openwhisk-deploy-kube/blob/0bc2e16478f5bceffc717a82ce581f82a2549344/README.md#setting-up-kubernetes. (Accessed on 06/27/2018).

[23] incubator-openwhisk/metrics.md at ee2dd9719ff09818adffd3a02707ca24118b8afa · apache/incubator-openwhisk. https://github.com/apache/incubator-openwhisk/blob/ee2dd9719ff09818adffd3a02707ca24118b8afa/docs/metrics.md. (Accessed on 06/27/2018).

[24] Kubeless. http://kubeless.io/docs/architecture/. (Accessed on 03/18/2018).

[25] Kubeless. http://kubeless.io/docs/. (Accessed on 03/18/2018).

[26] Kubernetes cloud controller manager - kubernetes. https://kubernetes.io/docs/tasks/administer-cluster/running-cloud-controller/. (Accessed on 06/26/2018).

[27] Kubernetes components. https://kubernetes.io/docs/concepts/overview/components/. (Accessed on 02/02/2018).

[28] Openfaas. https://docs.openfaas.com/. (Accessed on 06/26/2018).

[29] openfaas/faas: Openfaas - serverless functions made simple for docker & kubernetes. https://github.com/openfaas/faas. (Accessed on 03/27/2018).

[30] Serverless by the numbers: 2018 report. https://serverless.com/blog/serverless-by-the-numbers-2018-data-report/. (Accessed on 06/26/2018).

[31] Serverless computing.pdf. https://www2.deloitte.com/content/dam/Deloitte/tr/Documents/technology-media-telecommunications/Serverless%20Computing.pdf. (Accessed on 06/27/2018).

[32] Source code organization and your development workflow :: Serverless functions for kubernetes. https://docs.fission.io/0.8.0/tutorial/developer-workflow/. (Accessed on 07/05/2018).

[33] Vmware + openfaas - one month in - vmware open source blog. https://blogs.vmware.com/opensource/2018/03/20/vmware-openfaas-alex-ellis/. (Accessed on 06/27/2018).

[34] Watchdog - openfaas. https://docs.openfaas.com/architecture/watchdog/. (Accessed on 05/27/2018).

[35] What is kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. (Accessed on 01/11/2017).

[36] Who is using microservices? `http://microservices.io/articles/whoisusingmicroservices.html`. (Accessed on 02/25/2018).

[37] Adzic, G., and Chatley, R. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), ACM, pp. 884–889.

[38] Anderson, C. Docker [software engineering]. *IEEE Software 32*, 3 (2015), 102–c3.

[39] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al. A view of cloud computing. *Communications of the ACM 53*, 4 (2010), 50–58.

[40] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[41] Baldini, I., Castro, P., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., and Suter, P. Cloud-native, event-based programming for mobile applications. In *Proceedings of the International Conference on Mobile Software Engineering and Systems* (2016), ACM, pp. 287–288.

[42] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *ACM SIGOPS operating systems review* (2003), vol. 37, ACM, pp. 164–177.

[43] Bellard, F. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (2005), pp. 41–46.

[44] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, omega, and kubernetes. *Queue 14*, 1 (2016), 10.

[45] Castro, P., Ishakian, V., Muthusamy, V., and Slominski, A. Serverless programming (function as a service). In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on* (2017), IEEE, pp. 2658–2659.

[46] Che, J., Shi, C., Yu, Y., and Lin, W. A synthetical performance evaluation of openvz, xen and kvm. In *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific* (2010), IEEE, pp. 587–594.

[47] Claburn, T. Lambda and serverless is one of the worst forms of proprietary lock-in we've ever seen in the history of humanity. https://www.theregister.co.uk/2017/11/06/coreos_kubernetes_v_world/, Nov 2017. (Accessed on 02/21/2018).

[48] Cockcroft, A. Evolution of business logic from monoliths through microservices, to functions. https://read.acloud.guru/ff464b95a44d, 2017. (Accessed on 01/11/2017).

[49] Creasy, R. J. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development 25*, 5 (1981), 483–490.

[50] Dmitry, N., and Manfred, S.-S. On micro-services architecture. *International Journal of Open Information Technologies 2*, 9 (2014).

[51] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036* (2016).

[52] Dua, R., Raja, A. R., and Kakadia, D. Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on* (2014), IEEE, pp. 610–614.

[53] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On* (2015), IEEE, pp. 171–172.

[54] Fowler, M., and Lewis, J. Microservices. https://martinfowler.com/articles/microservices.html. (Accessed on 02/09/2018).

[55] Fox, G. C., Ishakian, V., Muthusamy, V., and Slominski, A. Status of serverless computing and function-as-a-service (faas) in industry and research. *arXiv preprint arXiv:1708.08028* (2017).

[56] Garriga, M. Towards a taxonomy of microservices architectures. In *International Conference on Software Engineering and Formal Methods* (2017), Springer, pp. 203–218.

[57] Goldberg, R. P. Architectural principles for virtual computer systems. Tech. rep., HARVARD UNIV CAMBRIDGE MA DIV OF ENGINEERING AND APPLIED PHYSICS, 1973.

[58] Honeycutt, J. Microsoft virtual pc 2004 technical overview. *Microsoft, Nov* (2003).

[59] Jaramillo, D., Nguyen, D. V., and Smart, R. Leveraging microservices architecture by using docker technology. In *SoutheastCon, 2016* (2016), IEEE, pp. 1–5.

[60] Javed, A., et al. Container-based iot sensor node on raspberry pi and the kubernetes cluster framework.

[61] Joy, A. M. Performance comparison between linux containers and virtual machines. In *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in* (2015), IEEE, pp. 342–346.

[62] Marston, S., Li, Z., Bandyopadhyay, S., Zhang, J., and Ghalsasi, A. Cloud computing?the business perspective. *Decision support systems 51*, 1 (2011), 176–189.

[63] McGrath, G., and Brenner, P. R. Serverless computing: Design, implementation, and performance. In *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on* (2017), IEEE, pp. 405–410.

[64] Morabito, R., Kjällman, J., and Komu, M. Hypervisors vs. lightweight virtualization: a performance comparison. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on* (2015), IEEE, pp. 386–393.

[65] Morabito, R., Petrolo, R., Loscrí, V., and Mitton, N. Enabling a lightweight edge gateway-as-a-service for the internet of things. In *Network of the Future (NOF), 2016 7th International Conference on the* (2016), IEEE, pp. 1–5.

[66] Muller, A., and Wilson, S. Virtualization with vmware esx server.

[67] Nastic, S., Rausch, T., Scekic, O., Dustdar, S., Gusev, M., Koteska, B., Kostoska, M., Jakimovski, B., Ristov, S., and Prodan, R. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing 21*, 4 (2017), 64–71.

[68] Ongaro, D., and Ousterhout, J. K. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference* (2014), pp. 305–319.

[69] Popek, G. J., and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM 17*, 7 (1974), 412–421.

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

[71] Rensin, D. K. Kubernetes-scheduling the future at cloud scale.

[72] Reshetova, E., Karhunen, J., Nyman, T., and Asokan, N. Security of os-level virtualization technologies. In *Nordic Conference on Secure IT Systems* (2014), Springer, pp. 77–93.

[73] Robin, J. S., Irvine, C. E., et al. Analysis of the intel pentium's ability to support a secure virtual machine monitor. Proceedings of the 9th USENIX Security Symposium, Denver, CO.

[74] Shillaker, S. A provider-friendly serverless framework for latency-critical applications. `http://conferences.inf.ed.ac.uk/EuroDW2018/papers/eurodw18-Shillaker.pdf`. (Accessed on 06/30/2018).

[75] Singleton, A. The economics of microservices. *IEEE Cloud Computing 3*, 5 (2016), 16–20.

[76] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 275–287.

[77] Sugerman, J., Venkitachalam, G., and Lim, B.-H. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track* (2001), pp. 1–14.

[78] Velte, A., and Velte, T. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[79] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at google

with borg. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 18.

[80] Xavier, M. G., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T., and De Rose, C. A. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on* (2013), IEEE, pp. 233–240.

# Appendix A

# Fission

## A.1 Function code

```
package main

import (
  "fmt"
  "net/http"
)

// Handler is the entry point for this fission function
func Handler(w http.ResponseWriter, r *http.Request) {
  body, _ := ioutil.ReadAll(r.Body)
  message := fmt.Sprintf("Hello world, input was: %s", string(
      body))
  w.Write([]byte(message))
}
```

## A.2 Examples of Commands

Below script creates a new go environment.

*$ fission env create --name go --image fission/go-env:0.8.0 –builderfission/go-builder:0.8.0*

Below script creates a new function using the environment created above.

*$ fission function create --name hello --env go --src hello.go --entrypoint Handler --minscale 5 –maxscale 5 –executortype newdeploy*

# Appendix B

# Kubeless

## B.1 Function code

```
package kubeless

import (
  "fmt"

  "github.com/kubeless/kubeless/pkg/functions"
)

func Handler(event functions.Event, context functions.Context)
    (string, error) {
  var err error
  message := fmt.Sprintf("Hello world, input was: %s", event.
     Data)
  return message, err
}
```

## B.2 Examples of Commands

Below script creates a new function.

*$ kubeless function deploy hello --runtime go1.10 --handler hello.Handler --from-file hello.go*

# Appendix C

# OpenFaaS

## C.1 Create a new function

The below script is used to create a new function environment. In the below script, faas-cli is the command, helloworld is the name of the function and go is the language. In OpenFaaS, the language is defined by a template. User can build their own custom templates and give it a custom name. OpenFaaS also has a predefined set of templates [9].

*$ faas-cli new helloworld --lang go*

### C.1.1 Auto generated function code

Below is an example of a code generated by use of the above script.

```
package main

import "fmt"

func main() {
    fmt.Println("Hello World!")
}
```

### C.1.2 Sample yml file for OpenFaaS functions

```
provider:
  name: faas
  gateway: http://172.42.42.102:31112/

functions:
  fun1:
    lang: go-streaming
    handler: ./fun1
    image: images.example.com/fun1
    secrets:
    - image-reg
    environment:
      write_debug: false
      read_debug: false
    labels:
      "com.openfaas.scale.min": "1"
      "com.openfaas.scale.max": "100"
functions:
  fun2:
    lang: go
    handler: ./fun2
    image: images.example.com/fun2
    secrets:
    - image-reg
    environment:
      write_debug: false
      read_debug: false
    labels:
      "com.openfaas.scale.min": "1"
      "com.openfaas.scale.max": "100"
```

## C.2 Deploy function

Functions in OpenFaaS are deployed by using the below three commands.

The build command builds the Docker image in the local system or the CI build server as may be the case.

*$ faas-cli build -f template.yml*

The push command pushes the Docker image to the image repository.

*$ faas-cli push -f template.yml*

The deploy command deploys the image as a function.

*$ faas-cli deploy -f template.yml*

# Appendix D

# Function Code

## D.1 Serializing Function - Classic

```
package function

import (
  "fmt"
)

// Handle a serverless request
func Handle(req []byte) string {
  return fmt.Sprintf("Hello, Go. You said: %s", string(req))
}
```

## D.2 HTTP Function

```
package function

import (
  "fmt"
  "net/http"

  "github.com/openfaas-incubator/go-function-sdk"
)

// Handle a function invocation
func Handle(req handler.Request) (handler.Response, error) {
  var err error

  message := fmt.Sprintf("Hello, Go. You said: %s", string(req
      .Body))
```

```
    return handler.Response{
        Body:       []byte(message),
        StatusCode: http.StatusOK,
    }, err
}
```

## D.3 Streaming Function

```
package function

import (
    "fmt"
)

// Handle a serverless request
func Handle(req []byte) string {
    return fmt.Sprintf("Hello, Go. You said: %s", string(req))
}
```