Date of acceptance Grade

Instructor Miika Komu

Serverless Computing on Constrained Edge Devices

Jan Tilles

Helsinki April 14, 2020

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Jan Tilles

Työn nimi — Arbetets titel — Title

Serverless Computing on Constrained Edge Devices

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	April 14, 2020	45 pages + 0 appendix pages

Tiivistelmä — Referat — Abstract

Serverless Computing aka Function as a Service is a cloud service model in which cloud provider manages computing resources and tenants deploy their code without knowing the details behind the underlying infrastructure.

The promise of serverless is to drive the costs down so that a tenant pays only for the computing resources that it actually utilizes instead of paying for idle containers or virtual machines. In this thesis, we discuss that Serverless Computing does not always fulfill these requirements. For instance, some serverless frameworks keep certain resources, such as containers or functions, idle in order to reduce latency during function invocation. This may be particularly problematic in edge domains where computing power and resources are limited.

In Function as a Service, the smallest unit of deployment is a function. These functions can be used, for example, to deploy traditional microservice-based applications. Serverless computing allows a tenant to run and scale functions with high availability. Serverless Computing also includes some tradeoffs: developers does not have so much of control over the underlying environment, testing of serverless functions is cumbersome, and commercial cloud service providers have a high degree of lock-in in their serverless technologies.

A serverless application is stateless by its nature, and it runs in a stateless container that is event-triggered and managed by the cloud provider. A serverless application can access databases but, in general, state related to the function itself is not stored in files or databases.

A number of commercial offerings and a wide range of open-source serverless frameworks are available. In this thesis, we present an overview of the different alternatives and show a qualitative comparison. We also show our benchmarking results with OpenFaaS running on an Kubernetes edge cloud (Raspberry Pi) based on algorithms typically utilized in machine learning.

ACM Computing Classification System (CCS):
General and reference → Document types → Surveys and overviews
Applied computing → Document management and text processing → Document management → Text editing

Avainsanat — Nyckelord — Keywords
Serverless Computing, Cloud Computing, Edge Computing, Function-as-a-Service, FaaS

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Supervisor Sasu Tarkoma

Acknowledgement

I would first like to thank my thesis instructor Miika Komu Senior Researcher at Ericsson. His useful remarks and engagement through the writing process of this master thesis was crucial and invaluable. I would like to express my gratitude to my supervisor, PhD Sasu Tarkoma of the Department of Computer Science at University of Helsinki.

I would also like to thank the experts at Ericsson who were involved in this research project: Jan Melen, Tomas Mecklin, Jimmy Kjällman, Joel Reijonen, Miljenko Opsenica, and Tero Kauppinen. Whenever I ran into a technical difficulties with my testbed or had a question about my research topics I could always count on their help.

I would like to thank my family and friends, who have supported me throughout this long journey of studies and writing of this master thesis. After all this has been a team effort and I will be grateful forever for you help. Thank you.

Author

Jan Tilles

Contents

1 Introduction

Function as a Service (FaaS) and Serverless Computing are new and emerging trends in cloud computing. Public cloud vendors have their own serverless computing offerings such as: IBM OpenWhisk, AWS Lambda, Microsoft Azure Functions and Google Cloud Functions. In the FaaS service model, developers can focus on the functionality of the application, while cloud platform providers are responsible for deploying, running and scaling their code. A serverless function is a piece of code that is executed in response to triggered events. An event that triggers the function can be, for example, HTTP call to API endpoint or message in a message queue. Serverless functions are reusable event-driven and short-lived procedures that do not store their own state. Thus, serverless functions are often called 'stateless'. However, the amount of serverless frameworks that provide access to databases and other persistent storage is gradually increasing.

Edge computing is another hot topic in a today's computer science. The paradigm of edge computing shifts computing utilities towards the edge of networks (e.g., closer to the end-user and IoT devices) in order to reduce limitations related to network bandwidth and especially latency. In the future, an edge device can be any type of device with an Internet connection and sufficient amount of computing power. As edge devices do not possess the computational capabilities of the datacenters, the edge computing platforms may vary a lot, ranging from Linux-based two unit server racks (located e.g., in a mobile base station) to small IoT devices.

The need of low latency and data-intensive applications empowered by IOT and mobile devices as well as small data centers at the edge of network challenge the traditional cloud computing. The resource limitations of the edge nodes require efficient usage of computing technologies that saves the resources of such devices. Running multiple processes, for example, multiple application in containers, might need optimization of resources in edge devices. In this thesis, we study a serverless platforms for the edge usage and evaluate if they could be used efficiently in constrained devices to save their limited resources.

In the recent years, a wide range of open-source serverless frameworks have emerged. For instance, some of these frameworks are evaluated [1][2] with Google Kubernetes Engine (GKE). Since the most popular open-source frameworks are running on the top of the Kubernetes container-orchestration system which supports largely x86 processor architecture, the current studies have not covered serverless framework evaluation on ARM-based devices. The lack of ARM support limits the use of serverless in application deployment on constrained devices such as the Raspberry Pi.

Serverless computing has also been utilized with machine learning (ML) [3][4][5]. The results have shown that serverless technology is particularly useful for the use cases of prediction and hyperparameter optimization. On the other hand, it has not yet redeemed its promise in processing and memory hungry model training and data pre-processing in ML. In particular, deep learning that applies neural networks to

text and natural language processing has been proven to be inefficient with serverless computing [6].

In this work, we attempt to create a scenario with a small edge system serving a simple prediction algorithm. We provide a comparison between OpenFaas serverless framework solution and Kubernetes Service to implement the prediction algorithm, and evaluate the performance of these solutions. Our intention is not to study machine learning as such, but to focus on how much additional overhead a serverless framework adds to a system with limited computational capabilities. We utilize a small Raspberry Pi cluster to run Kubernetes on top OpenFaas.

The rest of the thesis is organized as follows. Chapter 2 provides background information about the technologies, paradigms and concepts relevant to this thesis. Chapter 3 introduces Serverless Computing paradigm and five popular open-source serverless frameworks. Chapter 4 reviews the related work. Chapter 5 describes the design and implementation, and Chapter 6 shows a performance evaluation of the implementation. Finally, the two last chapters provide discussion and concluding remarks.

2 Background

The chapter provides background information on the technologies, paradigms and concepts relevant to this thesis. Sections 2.1 introduces some relevant concepts in cloud computing, with section 2.2 paying special attention to the edge computing. Section 2.3 describes the key concepts in cloud-based virtualization and also the microservice paradigm. At the end of this chapter, we introduce container orchestration, IoT and single-board computers.

2.1 Cloud Computing

Cloud computing is popular computer science paradigm where computing resources, data storage and services are outsourced to third party providers and made available as commodities. These third party providers are usually referred as Cloud Service Provider (CSP). Thus, cloud computing as such relies firmly on centralized service provisioning based on the client-server architecture. There are many definitions for cloud computing, among them, Rimal et al. [7] defines cloud computing as, *a model of service delivery and access where dynamically scalable and virtualized resources are provided as a service over the Internet.*Cloud computing is a focal point for the new technologies and paradigms such as mobile Internet, fog and edge computing, the Internet of Things (IoT), machine learning (ML), artificial intelligence (AI) and big data. In particular, the evolution of virtualization technologies has been driving the development and popularity of cloud services over the past decade. Virtualization enables the creation of multiple simulated environments or dedicated resources from a single physical device. We will go into more detail about virtualization in the

chapter 2.3.

There are two types of cloud service application: legacy and Cloud Native Application (CNA). Legacy application has traditionally been run on users machines or in companies own data centers. Example of such software are Microsoft Office and SAP, both now available as a cloud service. Cloud Native Applications are typically applications that based service-oriented architecture (SOA) implemented with, e.g., Linux containers, virtual machines (VM) or unikernels. Cloud Native Applications are typically horizontally scalable. Horizontal scaling is done automatically or on demand when application requires more resources. Scaling guarantee high availability (HA) of the application by scale out more containers.

2.1.1 Cloud deployment models

Cloud deployment models can be categorized into three main classes: public, private and hybrid clouds.

- **Public cloud** is owned and operated by a CSP. All hardware, software and infrastructure is owned by a service provider, who then delivers their computing resources to the customers over the Internet as a service. Public clouds are typically operated and accessed over web browser or SSH. Public cloud is multi-tenant by nature. Multi-tenancy is a architecture in which a single instance of a software application or resource (memory, CPU, storage) serves multiple customers. In cloud computing customer and customer organizations are called a tenants.

- **Private cloud** typically refers to a deployment owned and operated by a single business or organization, albeit new types of private clouds have emerged where companies pay to the third-party service providers to host their private cloud. For example, AWS Outpost and Azure Stack are Amazon's and Microsoft's offerings that bring cloud services and infrastructure on the premises. A private cloud gives tenant an opportunity to use cloud services without storing their data into multi-tenant public clouds. Data isolation is always a risk in multi-tenant public clouds. A private cloud is a one way of isolate the sensitive date from other tenants.

- **Hybrid cloud** is a combination of a public and private cloud interconnected together with technology that allows data and applications to be shared between different cloud deployments. A hybrid cloud environment can consist multiple public and private cloud providers.

A tenant is a group of users, company or organization sharing a common access to a specific service. A Public and hybrid clouds are multi-tenant by their nature. Multi-tenancy means that the infrastructure is shared by multiple tenants.

2.1.2 Cloud service models

Cloud computing utilizes a number of service models to provide computing utilities as services. The service provisioning is divided in three basic models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). On addition to these three traditional cloud computing service models, Function as a Service (FaaS) has gained popularity in recent years. In the different cloud service models, each layer abstracts the underlying service model (Figure 1).

- **Infrastructure as a Service (IaaS)** provides the underlying resources as physical or virtual computing resources to the tenant, including storage, network, firewall, load balancing, data partitioning, scaling, security, and data backup systems. In the IaaS model, a CSP typically gives the tenant the permission to allocate certain quota of resources from the cloud. Amazon AWS, Google Compute Engine, Microsoft Azure and IBM SmartCloud Enterprise have many technologies and services for building underlying infrastructure of private, public and hybrid clouds.

- **Platform as a Service (PaaS)** delivers a computing platform to tenant. Typically the delivered resources are programming-language runtime, database and web server. With PaaS, application developers can deploy, run and manage applications, for instance, based on microservice architectures, without worrying about the maintenance of the underlying infrastructure. Examples of PaaS services are Azure App Services[1], AWS Elastic Beanstalk[2] and Azure Search[3].

- **Software as a Service (SaaS)** is a cloud service model where tenant gain direct access to different "black-box" services. For instance, Database as a Service is typical SaaS service. In turn, the CSP scales the services on demand, and maintains the infrastructure and platform. Some of the well known SaaS services include Gmail, Google Drive and Google Calendar.

- **Function as a Service (FaaS)** is a new and quickly emerging trend in cloud computing. A FaaS system allows the developers to focus on application development using the provided APIs and to deploy software in smaller units, i.e, individual programming functions. The infrastructure is abstracted out similarly as in PaaS, but the main difference is that the billing can be more fine grained in FaaS (i.e. per function invocation) and the FaaS tenant pays for the real usage of service rather than a fixed monthly price. Three the most popular and well known FaaS platforms are: Google Functions, AWS Lambda and Microsoft Azure Functions.

[1] https://azure.microsoft.com/en-us/services/app-service/
[2] https://aws.amazon.com/elasticbeanstalk/
[3] https://azure.microsoft.com/en-us/services/search/

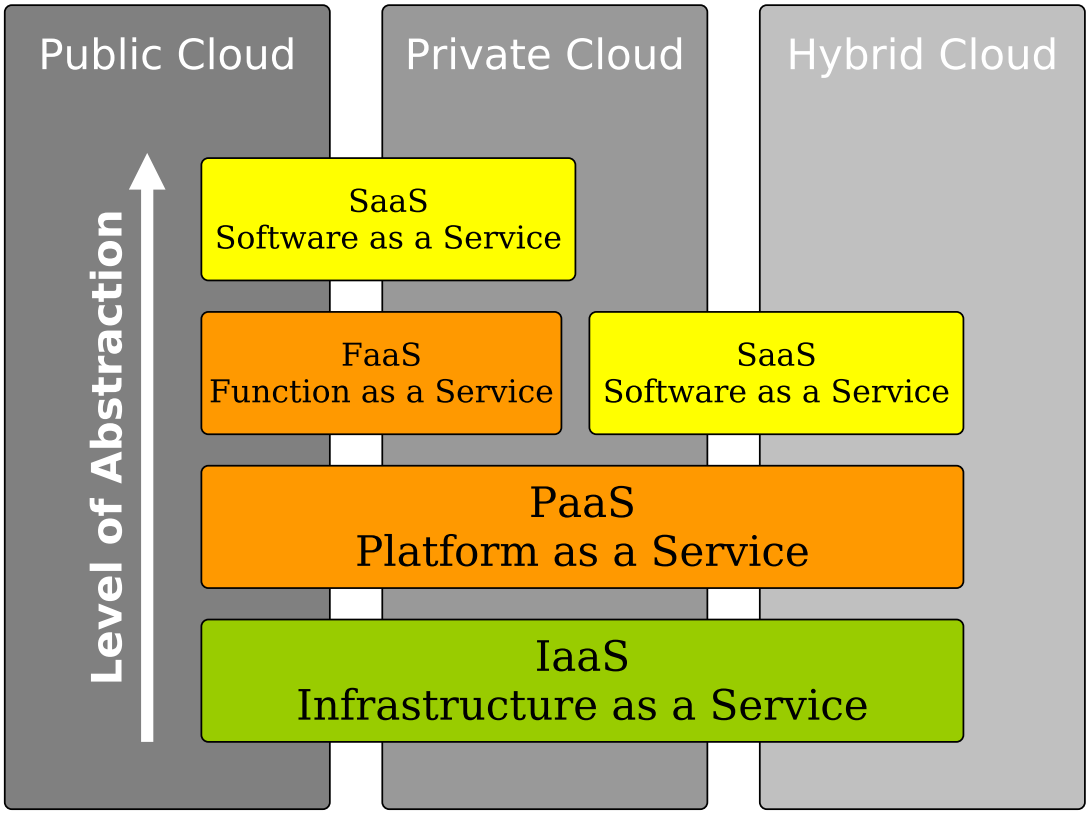

Figure 1: Cloud Service Model abstraction

As mentioned, Platform as a Service offers many of the same features as Function as a Service. Both eliminate the need for server hardware and software management. In PaaS, an application is deployed as a single unit, whereas a FaaS-based application is divided into distinct autonomous functions. A major difference in these two service models is the scalability of the application. A PaaS platform handles the scaling at the application level by running multiple instances of the application, whereas in FaaS each function is hosted separately, and load balancing is achieved by replicating individual functions.

In the IaaS billing model, the tenant is typically paying from all allocated resources that may often be idle. In the FaaS computing cost model, the tenant is billed based on the realized usage of service. In contrast, FaaS, also know as Serverless computing, is said to offer way of cutting down cloud computing costs due to fine-grained billing. Some studies have shown that complex pricing models [8] of serverless platforms of some cloud vendors and uneven request distribution [9] may be more expensive than using IaaS platforms for service deployment. Additional costs may incur in cases where serverless function utilizes other commercial services or transfers data though messaging services. For example, if AWS Lambda function reads or writes data to or from Amazon S3 Simple Cloud Storage Service[4] tenant is

[4]https://aws.amazon.com/s3/

billed for the read/write requests. Also, FaaS is stateless by its nature, and therefore it is not always a good fit for implementing a stateful service. FaaS services can also incur some extra latency when compared to IaaS because the function may not be ready serve and has to be instantiated. Nevertheless, it can complement the other service models when seem fit.

2.2 Edge Computing

Traditional cloud-based computing has two distinct limitations in terms of latency and network bandwidth: for instance, the end-to-end latency can be too high for some use cases, and network bandwidth to the Internet can be congested due to too many devices. Edge computing aims to tackle these limitations by moving computing utilities towards the edge of network, i.e., towards consumers of the edge services (e.g, end-users or IoT devices). An edge node can be any trusted computer or cluster that is connected to the Internet. Basically any vehicle, vessel, but even a mobile device or IoT device, could be used as a edge device as long the cloud can use it as a resource.

During the recent years, edge computing has become a popular alternative to the massive data centers operated in centralized clouds. When all or some computing utilities are moved closer to the actual sources of data, it can help developers to develop more latency and bandwidth optimized applications.

Popular terms, such as micro-data centers, intelligent edges, cloudlets, and fog, have been used interchangeably to describe edge computing in the literature. Cisco has defined the cloud extension concept of fog computing [10]. A fog node can be any industrial gateway, router or other device with the necessary processing power, storage capabilities and network connection [11]. Fog is a layer between cloud and edge, which extends cloud closer to the edge nodes that produce and act on IoT data. The Fog can be used to pre-process and analyze data nearby the originating edge nodes. Fog computing and edge computing have the same purpose, that is, to minimize latency and to reduce bandwidth usage.

Mobile-Edge Computing (MEC) is also part of the edge computing paradigm. In 2014, the European Telecommunications Standards Institute (ENSI) released an industry specification for mobile-edge computing with the following definition: *Mobile-edge Computing transforms base stations (e.g., 2G/3G/4G/5G) into intelligent service hubs that are capable of delivering highly personalized services directly from the very edge of the network within the radio access network (RAN) while providing the best possible performance in mobile networks* [12].

2.2.1 Edge types

Edge computing deployment models can be categorized into three main classes. Depending on the literature and bibliography, terminology can vary a lot. In the industry, there is a tendency to refer to regional clouds, localized data centre and

local devices.

- **Compute Edge** refers to a small datacenter located closer to the end-user and the data source than in the case of a centralized cloud data center. A compute edge has notably more processing and storage capabilities that end-devices and sensors closer to actual edge of the network. In contrast, high performance regional clouds and compute edges are not always located right next to the users, data sources or IoT devices, so their latency is usually larger than with compute edge.

- **Device Edge** consists of one or more small devices or servers. These nano scale data centers provide minimal processing and storage capabilities and can be deployed to environments like factories, cars and vessels. Device edge build with couple of servers in factory or residence can handle, for example, cooling, power consumption and heating of the building or factory. IoT devices benefit from the device edge the most. Since device edge is located in the near proximity of IoT sensors, issues related to latency and bandwidth are minimal.

- **Sensor Edge** consists IoT sensors, mobile devices or other small gadgets that are capable of collecting data and possibly able to control other physical devices like security cameras, house lighting or wearable like clothing or watches. Typically sensor edge devices utilize the cloud to storage their data and in some cases devices can managed directly from the cloud.

Small, single-board computing units such as Raspberry Pi, Arduino Mega 2560 and Onion Omega2Plus have narrowed the gap between device and sensor edge. Since these units have their own CPU, memory and data storage, and they are inexpensive to tweak with add-ons, sensor and camera units, it is sometimes hard to draw the line between device and sensor edge.

2.2.2 Mobile Edge Computing

One notable and developing edge computing area is Mobile Edge Computing (MEC) [13]. Mobile Edge Computing is vital part of the Edge Computing paradigm and its purpose is to extend cloud computing services to the edge of the network using mobile base stations [14]. Mobile edge computing was standardized in 2015 by European Telecommunications Standards Institute (ETSI) and Industry Specification Group (ISG)[5]. Mobile Edge Computing has multiple definitions, among them, Yu et al [15] have defined MEC as follows: *"Mobile Edge Computing (MEC) is an emerging technology in the 5G era which enables the provisioning of the cloud and IT services within the close proximity of mobile subscribers"*.

Another relevant edge computing architecture for mobile devices is Mobile Cloud Computing (MCC). MCC is based on computational offloading architecture that

[5]Industry Specification Group

mitigates resource constraints in modern smart phones by using resources of remote data centers. When we compare it to MEC, MCC has more notable challenges such as high latency, limitations related to network bandwidth and possible low coverage of remote data centers [16]. In MCC, an end-device utilizes the nearest data center available, so latency is dependent on the physical proximity. Instead, in MEC a device uses the nearest cloud service-enabled mobile base station. MEC is envisioned to be an important part of cellular technologies of the future and for new breed of wireless services. [17]

Mobile Edge Computing functionality is envisioned to be included within Radio Access Network (RAN) [18], that is, the essential part of cellular network communication infrastructure. RAN handles the communication between mobile device (smartphone, tablet) and any base station. RAN is the part of the mobile telecommunication system that connects mobile devices to mobile operator networks. Because a MEC can cover a wide geographic area, it is useful for low latency dependent edge services such as augmented reality applications, games, machine learning, self-driving cars, big data processing and other compute hungry applications. To summarize, the key benefits of MEC is that it push cloud assets such as computing resources, storage and network to the edge of the mobile network.

2.3 Virtualization

In cloud computing, virtualization is the process of creating a virtual version of any computing resource. Virtualization involves specific software to create a virtual version of this resource. With virtualization techniques, multiple operating systems and applications can be run on the same physical machine. The virtualization itself is often used as a synonym with hardware virtualization. The virtualization delivering Infrastructure as a Service solutions for cloud computing.

In cloud computing, multitenancy on a single host is attained with two main, alternative virtualization techniques: hypervisor-based or container-based virtualization. Virtualization enables software applications to run on emulated hardware (hypervisor-based virtualization) or on virtual operating systems (container-based virtualization).

A virtual machine is an isolated environment with limited access to the underlying computing, memory, storage and network resources. Each VM running on the same server appears to be running on top of separate physical machine, but they all are supported by a single host machine. Every virtual machine on the same host machine can run their own operating system (linux, windows etc.) and each VM in turn, can run multiple applications.

Container-based virtualization, or also known as containerization or operating system virtualization, can be achieved with so called containers. A container is an alternative virtualization technique to a virtual machine. A container run-time creates an abstracted running environment on top of the OS kernel, allowing guest process to run within a container in isolation of the others [2].

2.3.1 Virtual Machines

A virtual Machine Monitor (VMM), often called as hypervisor, runs on top of physical hardware, exporting a hardware-level abstraction to one or more guest operating systems. A guest OS runs on a virtual machine (VM) and operates with the virtual hardware the same way as a physical one. VMM traps all privileged operations and handles communication between the guest OS and the underlying physical hardware [19]. In other words, a VMM controls how guest operating system uses the hardware resources.

VMM is responsible for the isolation between virtual machines running on the same device. Operation of one virtual machine does not affect the other VMs running on the same VMM. This way, multiple services can be run on the same platform without influencing each other. This way, basic cloud multitenancy can be reached and the services of multiple tenants can run be on same device. Security of the multi-tenancy is still challenging to reach, while tenants sharing the same computer hardware [20][21]. For example, in 2018 a bug was found in the Intel CPUs, that have the potential to leak otherwise secure data. The services can be modified, moved, scaled and deployed independently without affecting other services. Xen and VMware are popular hypervisors, to name a few.

2.3.2 Containers

The Docker framework has popularized the use of containers via its ease of use, and all major commercial and open-source cloud platforms currently utilize and support Docker.Docker container is a running environment of a docker image file. Docker image comprised of multiple layers and used to execute code in a Docker container. A Docker image is a complete, static and executable version of an application. At runtime, these images are called containers, and they isolate software from its environment. For instance, containers are nowadays used for continuous integration and continuous delivery (CI/CD) workflows.

Containers are a lightweight virtualization alternative for hypervisor-based virtualization in cloud environments. Container-based virtualization does not employ virtual machine monitors. Also, unlike with hypervisors, containers are sharing the same kernel, instead of implementing their own guest operating systems. Container based virtualization leverages two existing Linux kernel features: namespaces and control cgroups [22]. The cgroups isolates resource usage such as CPU, memory, disk I/O and networking between processes, and namespaces are used to partition kernel resources so that a group of processes can utilize only a certain set of resources.

A container is a standard unit of software for packaging the components along with their dependencies, environment variables, libraries and source code that are necessary to run the desired software. Because a container isolates the software from its environment, they can be run in many different types of devices.

2.3.3 Unikernels

Unikernels are quite a new, but promising, technology for application deployment in cloud platforms. Unikernels have a small footprint, providing deployment agility and portability [23]. Similar to containers, Unikernels are a lightweight alternative to deploy microservice-based applications [24]. In the Unikernel concept, the software is integrated with the kernel it is running on. The kernel itself includes only the minimum required system calls and the drivers to run the software. An unikernel is an executable program running in a a single address space [25]. An unikernel is a minimal but a complete virtual machine to run a single process.

With reduced kernel implementation and simple system requirements, unikernels run faster than traditional virtual machines [26]. Image size difference between these two can be the order of gigabytes in favor of unikernels:. The small size of an unikernel makes it boot faster and to consume less memory [23] than legacy VMs. Compared to containers, unikernels consume much less memory [26]. Because of the unikernel structure, they can only run a single process.

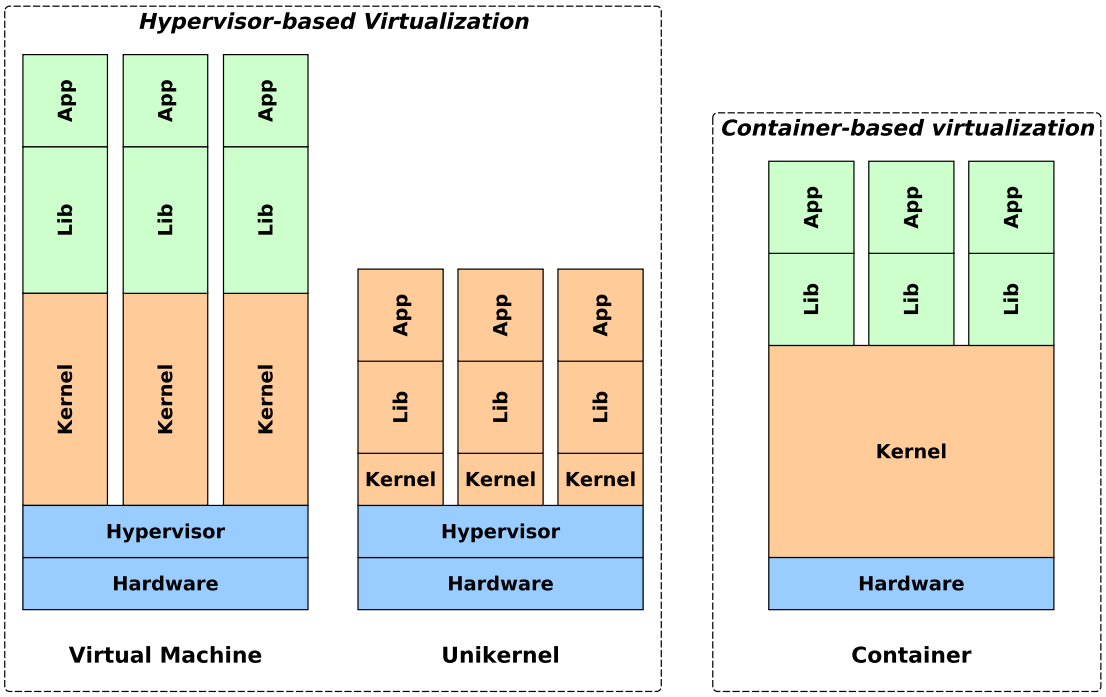

Figure 2: Comparison of virtual machine, unikernel and container system architecture

2.3.4 Microservices

Microservices are not a technology per se, but more of a software development technique, popularly used especially in the cloud environments. A microservice has

adopted the service-oriented architecture (SOA). SOA has emerged as an architectural style that provides loosely coupled, reusable and standard-based applications to integrate distributed systems [27]. For example, big monolithic applications can be broken down into smaller services using microservices. This approach allows better service isolation, and individual services can be built and deployed independently without impacting other microservices [28]. A microservice architecture can be implemented, for example, using the serverless techologies and/or containers.

2.4 Container orchestration

Containers have become a popular way to deploy services in modern software development. When services are hosted by a data center in production use, and thousands of containers can be running on a single cluster, it is necessary to have a stable container orchestration system. Without orchestration, applications would have to be deployed and scaled manually. A basic requirement of container orchestration is to efficiently deploy M containers into N compute resources. [29]

When developers deploy their applications and services in a cloud environment, it is common that these services are composed of multiple smaller microservices. When each microservice runs in a containerized environment, orchestration is needed to compose and coordinate the distributed services. Container orchestration and service definition have close relation. When application developer is responsible for defining the containers that are used to expose the service to outside world, the service definitions are used as input for orchestration. The Container Orchestrator operates with the resources that could be a VM or physical hardware [29]. In commercial setting, these VMs or physical devices are typically equipped with container-optimized operating systems, such as CoreOS, DCOS, and Atomic.

At the moment, the most popular container orchestration platforms are Kubernetes, Docker Swarm, and Mesos. Kubernetes originates from Google, and it is the de facto container orchestration software for containers. Swarm is a native orchestration solution for Linux containers published by Docker, and the Mesos is Apache's open source clustering software. In the following sections, we describe Kubernetes and Docker platforms more detail.

2.4.1 Kubernetes

Most container orchestrator platfoms can be deployed on almost any infrastructure. They can be deployed on physical hardware, VMs, or on almost any public or private cloud deployment. Kubernetes can also be deployed on a laptop. All major cloud vendors have their own services to run Kubernetes clusters, including Google Kubernetes Engine (Google), Azure Container Service (Microsoft Azure) and Amazon Elastic Container Service (AWS).

A Kubernetes cluster has three main components: master node, worker node and a distributed key-value store called etcd. Figure 3. shows Kubernetes architecture at

a high-level.

A master node is responsible for managing a Kubernetes cluster. The master node is the entry point for all administrative tasks in the cluster. Developers can communicate with the master node via the CLI, APIs or through Kubernetes UI (aka dashboard). Kubernetes allows multiple master nodes for a single cluster, but only one master at the time can orchestrate the worker nodes. If a cluster has multiple masters and the current master fails, a new master node is elected among the redundant master nodes. The main components of a Kubernetes master node are as follows:

- The API server located at the master node is responsible for the cluster administration. The API server validates and processes RESTful requests. After processing the requests, the resulting state of the cluster is stored in the distributed key-value store.
- Etcd[6] is a distributed key value store that is used for managing the state of the cluster. In Kubernetes, ectd is normally included as a part of the master node, but it can also be configured to another node.
- The Controller manages different non-terminating control loops, which regulate the state of the Kubernetes cluster. Each one of these control loops contains about the desired state of the objects it manages, and monitors their current state through the API server. In a control loop, if the current state of the objects the Controller manages does not meet the desired state, then the Controller takes corrective steps in order to make sure that the current state is the same as the desired state.
- The Scheduler distributes workloads to different worker nodes based on their type and available resources. Scheduler schedules the workloads as Pods or Services. (Figure 3)

A Pod is a basic scheduling unit in Kubernetes which consists of one or more containers. As multiple Pods may provide the same service redundantly, a Kubernetes Service can be used to group the Pods together and provide load balancing between them.

A worker node is a virtual machine, rack server or some other device, such as Raspberry Pi, that is capable of running Linux containers. All the worker nodes in a cluster are controlled by the master node. It is worth noting that when a service from Kubernetes cluster is exposed to the outside world, the clients are always communicating with the worker nodes, not with the master. Worker nodes have three main components as follows:

[6]https://coreos.com/etcd/

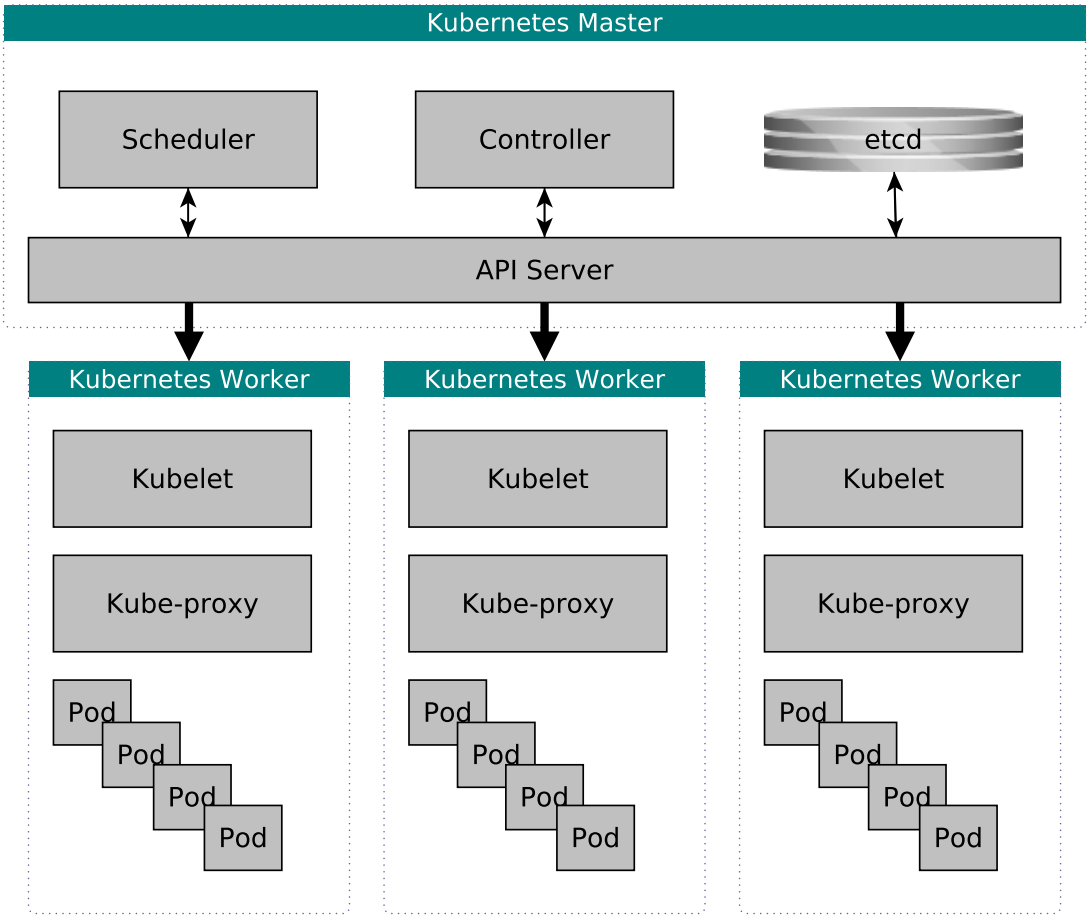

Figure 3: Kubernetes architecture

- The Container runtime is the software responsible for running containers. Kubernetes supports several runtimes: Docker, containerd, cri-o, rktlet and basically any basically implementation conforming to the Kubernetes CRI (Container Runtime Interface)[7]

- The Kubelet is an agent running on each worker node and communicating with the master node. It receives the Pod definition through the API server and runs the containers associated with the Pod. It also makes sure that the containers (belonging to certain Pod) are healthy at all times. The Kubelet interfaces with the container runtime using Container Runtime Interface (CRI)[8].

- The Kube-proxy is a network proxy running on every worker node. The Kube-proxy listens to the API server and routes incoming requests to corresponding service endpoints.

[7]https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md

[8]https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/

As we mentioned before, Kubernetes uses etcd to store the cluster state. Etcd is written in Go and uses the Raft[9] consensus algorithm to manage a highly-available replicated log [30]. Raft allows a collection of machines to work as a coherent group and makes the group fault-tolerant against failures of some of its members. Kubernetes also uses Etcd to store configuration details such as subnets, configmaps and secrets.

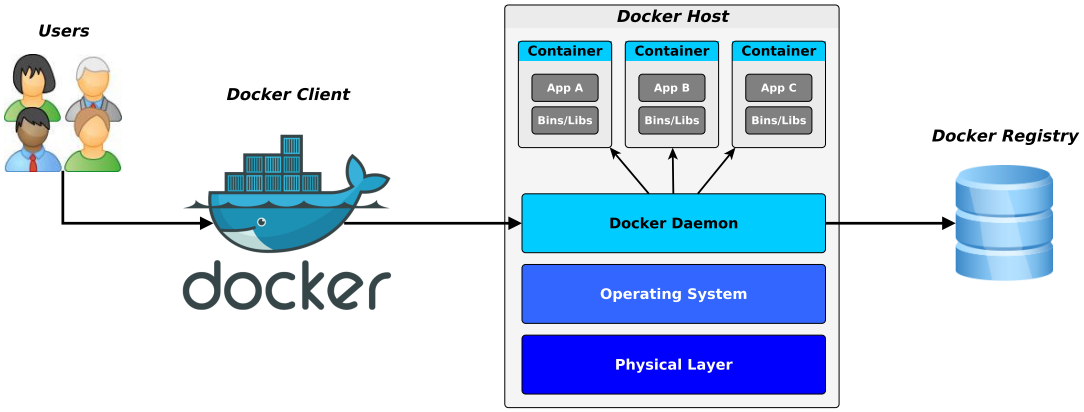

Figure 4: Docker architecture

2.4.2 Docker Swarm

Similarly as Kubernetes, Docker provides an alternative way to manage Linux Containers. Docker architecture is visualized in Figure 4. Docker is based on the classic client-server architecture. In the Docker architecture, the two main components are Docker *client* and Docker *daemon*. Docker daemon manages all the framework objects such as a images, containers, networks, and volumes. It is responsible for running, building and distributing Docker containers. Docker client and also the daemon uses a REST API for communication over UNIX sockets or alternatively a network interface. A daemon can also communicate with other daemons in order to manage Docker services in cluster setting.

The Docker client and daemon can run on the same system or different machines in the case of cluster. The Docker client is the primary way interact with the Docker environment and the client can communicate with more than one daemon. The third important part of the Docker architecture is a Docker registry that stores the docker images. Docker Hub is a public registry that anyone can use, but developers can as well run their own private registries on a local computer or an on-premise server. Docker is configured to look for images in the Docker Hub by default.

Figure 5 shows the general Docker Swarm architecture. Docker Swarm is a selection of multiple Docker nodes operated in swarm mode. A node in Docker Swarm can act as manager, worker or alternatively perform in both roles. A Swarm manager

[9]https://raft.github.io/

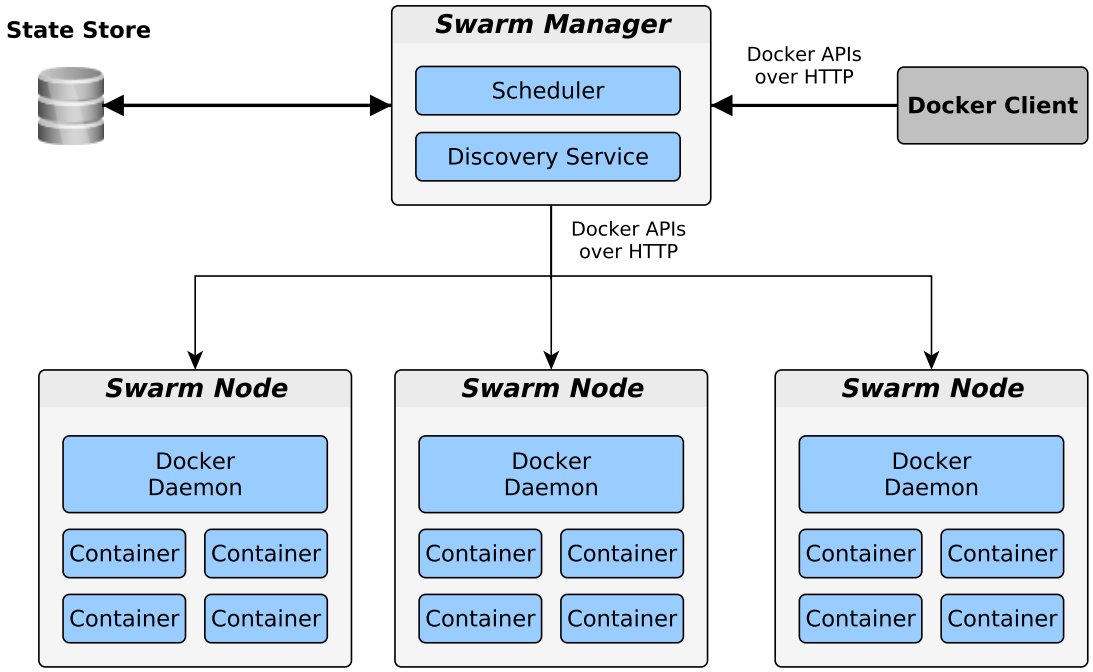

Figure 5: Docker Swarm architecture

manages membership and schedules tasks to worker nodes, with each task running in a single container. On the other hand, a worker is responsible for serving the workloads. In Docker Swarm environment, a developer defines an optimal state for a deployed service by configuring the number of replicas, network and storage resources. Docker Swarm is responsible of creating a the service matching the configurations. If the service needs to be updated, a developer updates the configurations and Docker Swarm for example stops outdated services and creates new ones. Swarm services are exposed to the outside world via ports.

2.5 IOT and single-board computers

The Internet of Things (IoT) can be defined as a network of networks. In the future, an IoT network connects possibly billions of Internet connected devices. IoT has lot of uses cases such as connected cars, wearables, sensors and robots. Many IoT devices are connected to Internet with wireless, short-range communication technologies such as ZigBee, Bluetooth and RFID [31]. Often IoT networks utilize the 2.4GHz and 5GHz WI-FI bands to gain fast communication speeds.

In this thesis, we utilize a an ARM-based single-board IOT device. Single-board computers are often credit card sized computers equipped with processor, memory and IO bus. Raspberry Pi[10] is one of the most popular single-board computers, and its popularity is based on the low price. At the time of writing, the latest

[10]https://www.raspberrypi.org/

version is the Raspberry Pi Model 4 with a 1.5GHz 64-bit quad-core processor, 1GB, 2GB or 4GB LPDDR4-3200 SDRAM (depending on model), Bluetooth 5.0/BLE, Gigabit Ethernet, and PoE (Power-over-Ethernet) support. Wide industrial interest in IoT and machine learning have attracted developer to use single-board computers in the recent years. Boards such as Google's Coral[11] and Nvidia's Jetson Nano[12] utilize TPU (Tensor Processing Unit) AI accelerator and GPU (Graphics Processing Unit) technologies in order to enable faster machine learning implementation. The TPU AI accelerator is designed for Google's TensorFlow[13] framework which is a symbolic math library for machine learning applications. GPU kernel performance for machine learning is widely studied [32][33][34][35].

3 Serverless Computing

Serverless computing, aka Function as a Service (FaaS), is an emerging trend in cloud computing. Serverless is a set of techniques and technologies abstracting away the underlying infrastructure and platform, including the operating system, programming-language runtime, database and web server. The term serverless itself is misleading because the serverless functions actually do run on some servers somewhere. In a serverless architecture, developers do not need to know anything about the underlying infrastructure, while the service provider is responsible for building, maintaining and scaling of these functions (Figure 6). Serverless computing can be defined as dynamically allocated resources for event-driven function execution [36].

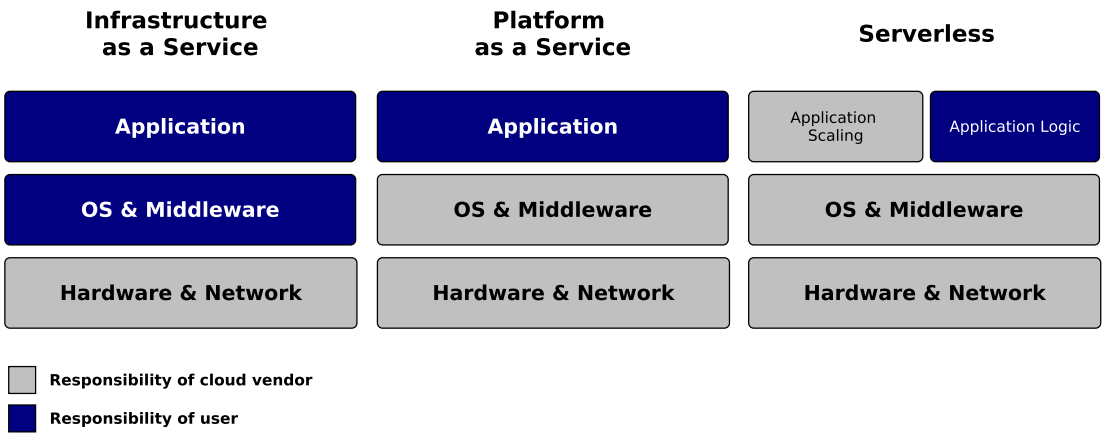

Figure 6: The key difference between Serverless Computing and common Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) models.

In serverless computing an application is split into individual small functions. Typically individual function performs a single task. For example, function can returns

[11] https://coral.withgoogle.com/
[12] https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/
[13] https://www.tensorflow.org/

value from database. In a FaaS system, all functions are short-lived and are invoked by events. To be more precise, an event can be, for example, an HTTP request, a message in a message queue or a system monitoring alert. Message queues used in a serverless frameworks are messaging technologies for cloud native applications, IoT messaging, and microservices architectures. Messaging technologies such as Nats [14], Kafka [15] and Kinesis [16] can be used in serverless frameworks. The result of the function execution can be an HTTP reply or a response message to the message queue [37]. Most of the existing FaaS implementations require the functions to be stateless; a serverless function does not keep its state information on a host file system or in a database.

Serverless computing brings significant advantages over traditional three-tier apps and Platform as a Service (PaaS) solutions, but also some disadvantages.

- **Agility**. Developing traditional monolithic applications and deploying three-tier apps in software rollouts are time consuming operations. An application that is developed based on a serverless architecture can enable more rapid deployments implementing new features with zero-maintenance.
- **Control**. With serverless application development, developers can concentrate on coding without worrying about server provisioning or scaling. Respectively, developers do not have so much of control over the environment because FaaS service provider defines the environment. Thus, the service provider is also liable for the maintenance and scalability capabilities of their services.
- **Cost**. In serverless computing, the customer, or tenant, only pays for the actual computation. Compared to virtual machines and containers, serverless enables multiple versions of a function without incurring the extra overhead of unused resources.
- **Testing**. The fine granularity of the developed components in serverless computing makes debugging and service troubleshooting more difficult. High fidelity also influences the tools and frameworks that are used in serverless computing [38].
- **Vendor lock-in**. Commercial cloud service providers have a high degree of lock-in in their serverless technologies.
- **Open Source**. Multiple open source serverless frameworks have emerge recent years. The frameworks try to mitigate vendor lock-in by using vendor neutral technologies such as Docker Swarm and Kubernetes. For example, open source frameworks, such as Kubeless, Fission and OpenFaaS, can utilize Docker and Kubernetes frameworks.

[14] https://nats.io/documentation/streaming/nats-streaming-intro/
[15] https://kafka.apache.org/
[16] https://aws.amazon.com/kinesis/

3.1 Serverless Frameworks

Cloud-native application developers have adopted Kubernetes widely and it has become the de facto standard for deploying containerized applications at scale. The success of Kubernetes has also influenced Serverless Frameworks. Most of the open-source frameworks such as Kubeless, Fission and OpenFaas are abstracting and utilizing underlying the Kubernetes architecture. Maturity of such frameworks is then bound to Kubernetes itself. However, some frameworks are independent from Kubernetes. Standalone Serverless frameworks include OpenWhisk and Nuclio which are not really kubenetes-native frameworks, but implement their own architecture. These two frameworks do not need Kubernetes to run Serverless Functions but are capable of using K8s load balancing for scaling if needed.

3.1.1 Kubeless

Kubeless is Kubernetes-native serverless framework that leverages the Kubernetes Custom Resource Definition [17] and Custom Controller[18] concepts to make functions as a custom resources in Kubernetes. Kubeless can build and store functions as docker images that can be stored in a docker registry. If function is built as a docker image, the same image can be used in every function deployment. This way, developers can ensure that each function is similar in each domain.

Kubernetes Custom Resources are extensions of the Kubernetes API, the endpoints that are not available in default Kubernetes installation. Kubeless uses Kubernetes Deployment and Pods to serve the functions. Kubeless also uses ConfigMap to insert function code into container runtimes. By default, each deployed Kubeless function is bound to a Kubernetes Service with virtual IP address. Kubeless functions are exposed publicly through Kubernetes Ingresses. Kubernetes Ingress is an API object managing external access, typically HTTP, to the cluster services.

A Kubeless deployment includes a Kubeless Function Controller that monitors container liveness and takes the necessary actions if needed. Monitoring in Kubeless is based on Prometheus data. Each Kubeless function is deployed to a separate Kubernetes deployment and Horizontal Pod Autoscaling (HPA) scale function based on the defined workload metrics. The Horizontal Pod Autoscaler is Kubernetes API resource that automatically scales the number of pods. For example, to autoscale based on CPU usage, the developer has to deploy the corresponding function with the CPU request limit set using the *–cpu* parameter.

For deploying new functions, a developer uses Kubeless CLI to produce the Function object (Figure 7). The object is submitted to the Kubernetes API server and The Kubeless Function Controller handles the deployment of the function using the Kubeless framework. When the Kubeless Function Controller detects a new Function object, it reads its content and creates a ConfigMap with the function content

[17] https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/
[18] https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

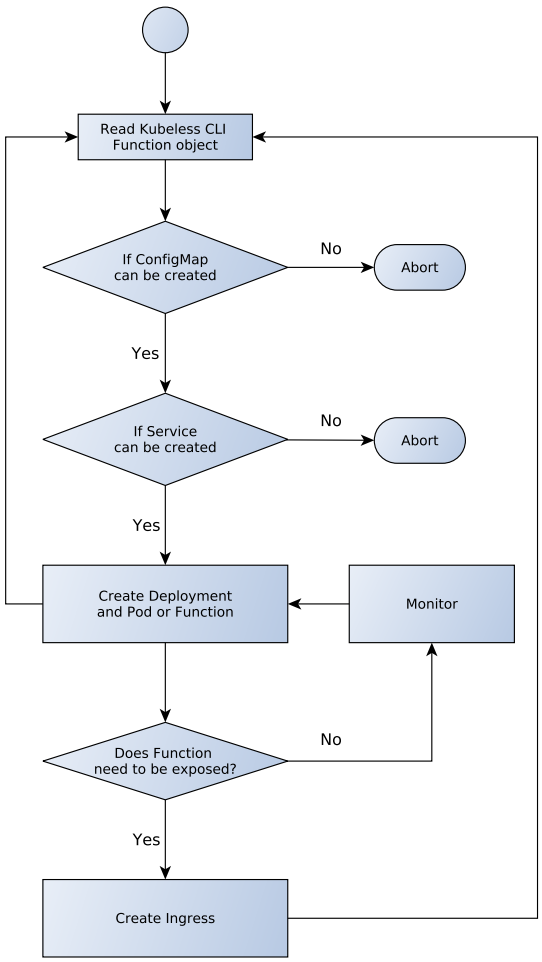

Figure 7: The Kubeless process to run a function.

and the all needed dependencies. Next, the framework creates a Service for the Function. If the controller fails to create a ConfigMap or a Service, it will not create the Deployment at all and the procedure will be cancelled. If a Deployment can be created, the Pod is generated with the function. Since a Kubeless functions are Custom Resources, the developer can access them by using kubectl, or example, by executing *kubectl get functions* to print out the deployed Kubeless functions.

After Kubeless has deployed the function, the framework makes it available with HTTP request via event-based or asynchronous publish-subscribe model or via scheduled events. Kubeless exposes publish-subscribe (pubsub) topics in StatefulSets of the Kubernetes using two open source frameworks, Kafka[19] and Zookeeper[20].

Kubeless monitoring relies on Prometheus[21]. In Kubeless, the language runtime automatically collects metrics for each function and the data is displayed in the default

[19]https://kafka.apache.org
[20]https://zookeeper.apache.org
[21]https://prometheus.io/

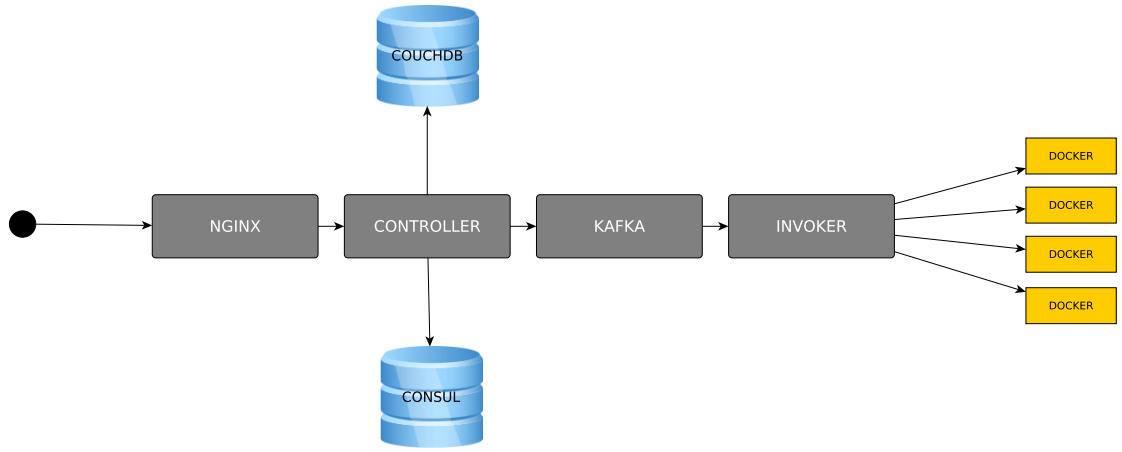

Figure 8: High-level architecture of OpenWhisk

Prometheus dashboard. Developers can also use Grafana[22] in order to visualize the Prometeus metrics.

At the moment, Kubeless supports two major Kubernetes versions, 1.8 and 1.9. Software runtime support in Kubeless includes python (2.7, 3.4, 3.6), nodejs (v6 and v8), ruby (2.3, 2.4, 2.5), php7.2, go1.10, dotnetcore2.0, java1.8 and ballerina0.981.0.

3.1.2 OpenWhisk

Apache OpenWhisk is an open source, serverless framework initially developed by the IBM. Later, OpenWhisk was submitted to Apache Incubator project which was then used in IBM's serverless computing offering for IBM Bluemix. The OpenWhisk programming model is based on three primitives: action, trigger and feeds [23].

In OpenWhisk, a function executing program code is called an action. Action is any code that is written by a developer using an programming language supported by OpenWhisk. These actions can be dynamically scheduled, ran and invoked using triggers. Events that cause the triggering are coming from external sources called feeds. The feed or event source can be for example database, message queue, web application, sensor, chatbot or scheduled task. OpenWhisk also supports HTTP requests to invoke actions.

Figure 8 introduces the main components of the OpenWhisk system architecture. Every request to OpenWhisk infrastructure goes through Nginx which is a web server acting as a reverse proxy for a Controller, and the purpose of the Nginx is to hide the actual implementation of the OpenWhisk API of the Controller. The Controller is responsible for authentication and authorization of every incoming request. The Controller decides the path that a request traverses on the OpenWhisk infrastructure.

[22]https://grafana.com/
[23]https://openwhisk.apache.org/

The state of the OpenWhisk system is stored in CouchDB. The definitions of actions, triggers and rules that associate triggers to a certain action are stored in CouchDB along with other metadata and credentials related to the OpenWhisk system. Architecture uses a distributed key-value store called Consul for state management. The Controller uses Consul for service discovery of the entities that will invoke an action. Invokers are responsible for execution of the code. After a Controller has authenticated the request, it will pass the control to the Invoker component by using Kafka. The Invoker handles the last step in the serverless function execution by launching a new Docker container. The invoker also copies the source code from CouchDB and injects the code to the associated container. The Invoker can reuse existing a hot, i.e., running, container, or restart a paused warm container, or launch a new cold container for a new invocation [2].

Currently, OpenWhisk supports programming languages such as NodeJS, Swift, Java, Go, Scala, Python, PHP, Ruby and Ballerina.

3.1.3 Fission

Fission is a open source project from Platform9 Systems. It is a Kubernetes-native serverless framework that does not have support for Docker Swarm. At the moment, Fission is supporting Python, NodeJS, Go, Ruby, C# and PHP programming languages, but developers can extend its language support by building own custom containers to execute programming language of their choice.

Fission has three key concepts: Function, Environment and Trigger:

- **Function** is a module with one entry point that Fission executes. Entry point is a Function with a certain interface written in supported programming language like Python or Ruby.
- **Environment** is the language-specific part of Fission that contains resources to build and run a Function. Fission invokes Functions with HTTP calls. In practice, the runtime of an Environment is a container with an HTTP server and a dynamic loader that loads a Function. Fission has pre-built environments for NodeJS, Python 3, Ruby, Go, .NET, .NET 2.0, Perl and PHP7.
- **Trigger** is the part of the framework that binds Events to Function invocations. In Fission, an event invokes a Function and Trigger configures Fission to use a specific Event to invoke a Function. It is also worth noting that the Fission framework supports several types of triggers. The main triggering method used in Fission is an HTTP Trigger. Besides HTTP Triggers, Fission supports Timer Triggers, open-source messaging system NATS, Azure Message Queue triggers, and also native Kubernetes Watch triggers.

In Fission framework, an executor controls how function pods are created and

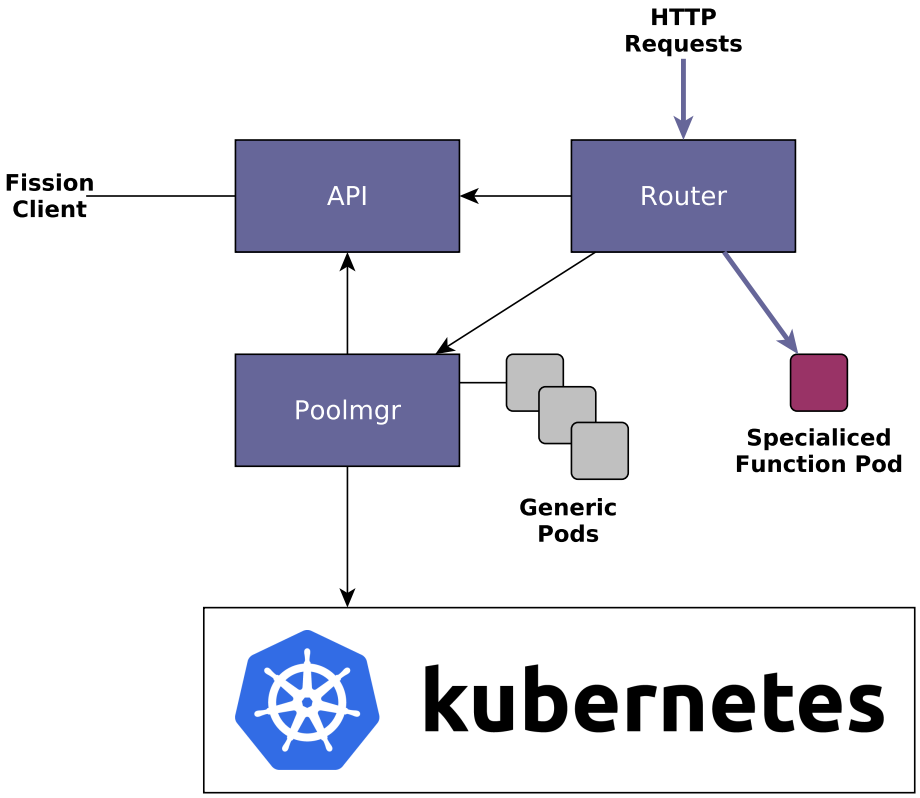

Figure 9: Fission pool-based executor architecture.

what capabilities are available. Fission supports pool-based (Figure 9) and new-deployment executors (Figure 10) as explained next.

A pool-based executor (Poolmgr) communicates with Kubernetes and creates a pool of generic pods, by default three pods per environment [37]. These containers have a dynamic loader for loading a function into a container when the function is invoked. When the function is created and invoked, one of the three pods is removed from ReplicaSet and used for execution. The same pod serves the subsequent requests for the function invocations, and later the pod is cleaned up when no requests are arriving. It is worth noting that Poolmgr execution is suitable for functions with low latency requirements but it does not support on-demand-based autoscaling.

A new-deployment executor (Newdeploy) utilizes Kubernetes infrastructure more heavily than Poolmgr. Newdeploy creates a Kubenetes Deployment along with a Service and HorisontalPodAutoscaler (HPA) to serve function execution [24]. With autoscaling enabled, Fission is able to autoscale function pods and handle load balancing between them. Newdeploy executor type is recommended to be used for requests without low-latency requirements and asynchronous function calls. During the first function call, Kubernetes deployment along with Service and HPA will be created. All subsequent requests can be served with same deployment. Idle objects

[24]https://docs.fission.io/concepts/executor/

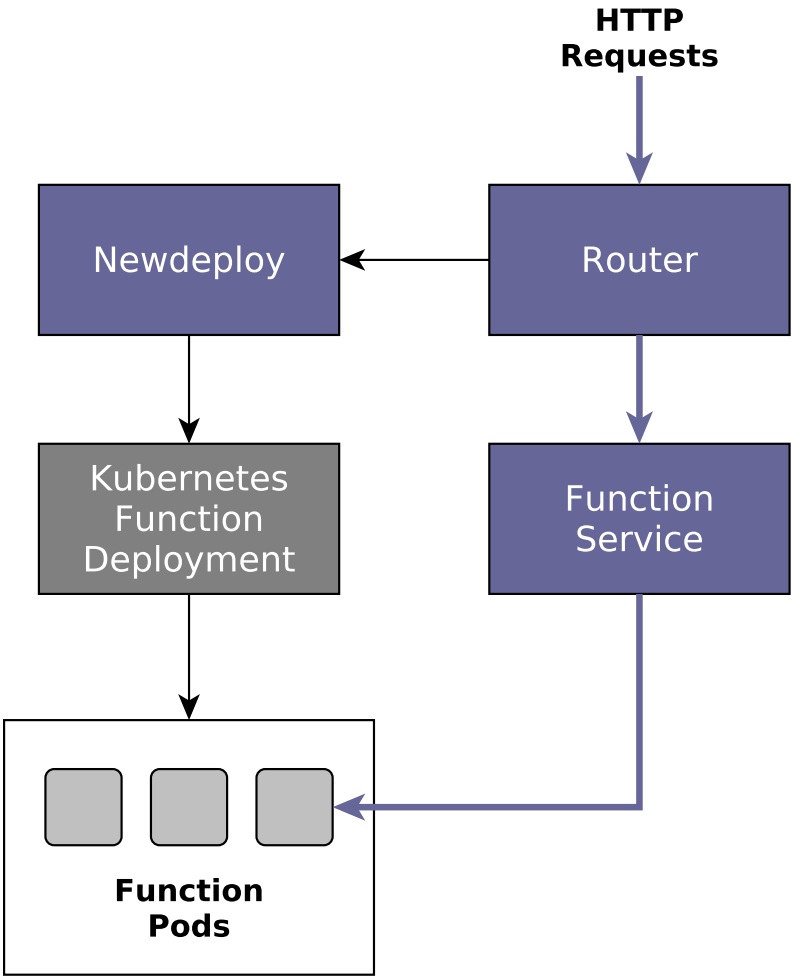

Figure 10: Fission Newdeploy executor architecture.

are cleaned up in a similar fashion as in the Poolmgr.

Newdeploy uses a "minscale" value of zero by default, but for low-latency applications this minscale value can be set to greater than zero. The idea is that Fission keeps minscale number of pods ready to serve a Function in order to reduce latency in function invocation. Minscale value ensures that pods stay idle but are not cleaned up. In the case latency needs to be prioritized at the expense of idle resources, Newdeploy execution type is recommended. It is worth noting that the Fission team is planning to add support for volumes for newdeploy in the future.

3.1.4 OpenFaaS

OpenFaaS® is an open source FaaS framework for building serverless functions with Docker and Kubernetes (Figure 11). OpenFaaS is written in Golang and is MIT licensed. In OpenFaaS, any software can be packaged as a function. A developer

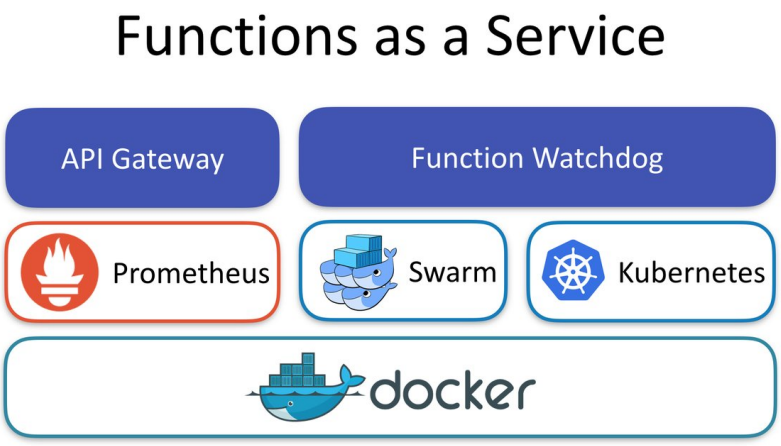

Figure 11: OpenFaaS system architecture.

can turn a Docker image into a serverless function by adding an OpenFaaS watchdog. OpenFaaS has built-in support for multiple programming languages through templating [25], and OpenFaaS gives the developer ability to write and use their own templates. The OpenFaaS framework has a build-in template engine which allows developer to create new functions in a given programming language. Template engine scaffold out function skeleton, that developer can reinforce with own function code.

API Gateway

The API Gateway collects Cloud Native metrics from open-source monitoring solution Prometheus and it is external route into deployed functions. The gateway is responsible of scaling functions according to the load and based on alerts received from Prometheus. The API gateway scales functions by altering replica count in Docker Swarm or Kubernetes. OpenFaaS functions are triggered using the OpenFaaS Gateway API.

Function Watchdog

The Function Watchdog (Figure 12) is an embedded HTTP server that supports concurrent requests. Watchdog is responsible for handling function timeouts and pod healthchecks in the OpenFaaS environment. Every OpenFaaS function can use a watchdog as a entrypoint which is the init process for the container running the function.

In OpenFaaS, watchdogs come in two flavors, "classic" and the newer "of-watchdog". Former is a standard version included in all OpenFaaS templates. The classic watchdog approach forks one process per request. After the process is forked, the watchdog passes a request to the function via stdin and reads the corresponding response via stdout. This approach with a single process forking has multiple advantages such as process isolation, portability and simplicity. The latter, of-watchdog, is more suit-

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

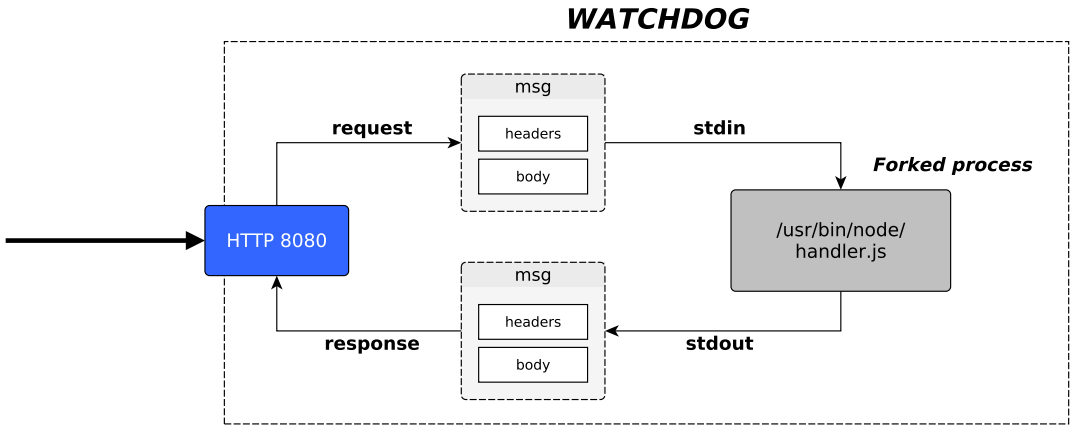

Figure 12: OpenFaaS Watchdog

able in streaming use-cases or in resource-intensive operations, such as serving data from databases or serving machine learning models. The Of-watchdog us based on "http mode" where processes can be re-used to avoid the latency of forking. Of-watchdog keeps a function process running between invocations.

Triggering

The most common way to trigger an OpenFaaS function is using a HTTP request. Similarly as with most of the other open-source frameworks, also OpenFaas utilizes NATS Streaming in order to invoke functions. NATS Streaming server used in NATS Streaming is a proxy server that decouples HTTP messaging between the caller and the function.

OpenFaaS is producing asynchronous and synchronous versions for every function. The asynchronous functions can be used for long-running tasks and function timeout is different than synchronous timeout at the gateway. When OpenFaaS is deployed on top of a Kubernetes cluster, functions can be run as Linux cron jobs using Cron Job resources. OpenFaaS is using connectors that allows developers to create a separate microservice to map functions into topics and then to invoke them via API Gateway. OpenFaas supports many different connectors such as Apache Kafka, AWS SNS, Minio, CloudEvents, IFTTT, VMware vCenter and Redis. Developers can also fork the OpenFaaS event connector SDK and create their own connectors between publisher-subscriber topics and OpenFaaS functions.

3.1.5 Nuclio

Nuclio is a Serverless project, derived from Iguazio [26], a Platform as a Service for data science. Nuclio functions can be run in standalone Docker containers or on top of a Kubernetes cluster. On top of that, Go or C based Function Processor can be compiled into a single binary. In the center of its high-level architecture is the

[26] https://www.iguazio.com/

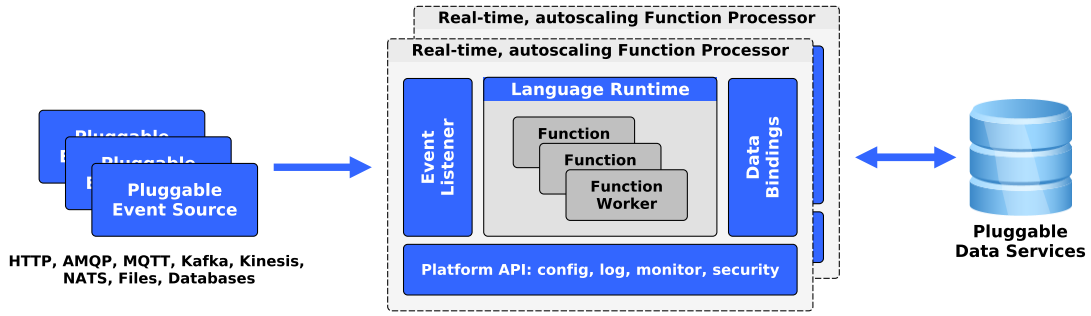

Figure 13: Nuclio's high-level architecture

Function Processor written in Go (Figure 13).

A Function Processor can listen to different types of triggers using Event Listeners, and it invokes function execution. Like most of the other Serverless Frameworks, Nuclio is also supporting HTTP, message queues (RabbitMQ, MQTT, NATS), different data streams (Kinesis, Kafka). Nuclio can listen to UNIX sockets to fetch data from external events and other data sources such as databases.

A Runtime Engine (Runtime) initializes the environment for functions by setting up variables, context, log and data bindings. A Runtime is also responsible for feeding event objects to Function Workers that execute function code. A Runtime is also responsible for returning responses to event sources.

Each Function Worker is able to process one message at a time. A Runtime can have more than one independent and parallel Function Workers which supports non-blocking operations and, at the same time, maximizes CPU utilization and supports concurrency within the same processor. Service autoscaling can be achieved, for example, by using Kubernetes to scale up the number of Function Processors.

Nuclio supports three types of processor Runtime implementations: Native, SHMEM and Shell. Native is for real-time routines written in Go and C. SHMEM can be used with programming languages, such as Python, Java, and Node.js. The processor communicates with the SHMEM function runtime via zero-copy shared-memory channels. Shell runtime can be used with command-line execution or executable binaries. Function Processor runs executables using a shell, with the required command and environment variables, and then adds the executable's standard-output (stdout) and standard-error (stderr) logs to the function results.

Nuclio functions can access external resources such as objects, files and databases using Data Binding interface. The Data Binding interface is responsible for connecting to resources, security and cache. To be more precise, functions that can read and write to local file system files, access databases and use TCP or HTTP to access remote data. The Data Binding unit enables this without changing the function code.

Control Framework initializes and controls processor components mentioned above. The framework provides logging capabilities for the Function Processor, monitors

function execution statistics and interacts with the underlying platform that can be Kubernetes, for instance.

A single Nuclio Function Processor can outperform most serverless/Faas solutions. Nuclio can run 300000 function invocations per second with a simple Go function and up to 70,000 events per second when using functions written with Python (PyPy) or node.js. Some studies have shown that Nuclio functions can respond in less than 0.1ms latency [39]. Nuclio framework is distributed in two versions: free open-source and enterprise grade versions. The open-source version lacks the most advanced features such as integration with machine learning tools and shared volume support between functions.

3.1.6 Comparison

All the five presented serverless frameworks include an open-source version licenced under Apache 2.0 or MIT. We have added AWS lambda in our comparison, because it is widely considered the de facto implementation of serverless technologies. Nuclio also offers a commercial hosted version of their framework that supports autoscaling. In contrast, this is supported by default by the others. OpenFaas alert manager and OpenWhisk autoscale based on number of requests. Fission and Kubeless autoscale based on CPU metrics, but Kubeless can also use customized autoscaling rules that a developer has defined. AWS lambda autoscale based on number of requests and function instances can reach between 500 and 3000 replicas. Functions can scale by an additional 500 instances, each minute and it continues until there are enough resources to serve all requests, or a replica limit is reached. When the number of requests decreases, Lambda stops unused instances.

Kubeless, OpenWhisk, Fission and Nuclio lack ARM support, whereas OpenFaaS supports ARM by using customized function templates. OpenFaaS architecture is not bound to certain programming language runtimes like the other frameworks, which give developers freedom to expand architecture by adding their own programming language templates. OpenFaas templates contain a Dockerfile which gives opportunity to build and package a function with any Docker image. AWS Greengrass can run AWS Lambda functions on constrained devices and have ARMv7l support for Raspberry Pi.

Nuclio is the only framework that allows running workloads on GPUs. Considering training for machine learning models this is an advantage in comparison to the other frameworks. All serverless frameworks support the most popular programming languages and the most common HTTP and event-based function triggering methods. Nuclio is again most diverse when comes to function triggers and message queue integration. Nuclio support currently 5 different message queue systems, such as Kafka, NATS, Kinesis, RabbitMQ and MQTT, whereas other frameworks rely solely on Kafka and NATS. When it comes to container orchestration, Kubeless and Fission are fully Kubernetes-native and do not support other container orchestrators. OpenWhisk and Nuclio do not require the use of any container orchestrator

	Local Docker	Image Repository	Base Image
OpenFaaS	Required	Required	Required
Fission	None	None	Required
OpenWhisk	None	None	Required
Kubeless	Required	Required	Required
Nuclio	Required	Required	Required

Table 1: Comparison of container image use in serverless frameworks.

but still offer support for Kubernetes. OpenFaas is the only framework supporting multiple container orchestrators: Kubernetes, Docker Swarm and Nomad.

One distinct difference between serverless frameworks is how they use container images and image registries (Table 1). For example OpenFaaS is fully based on the use of Docker. Functions are packaged as containers, built using the local Docker installation and function images are pushed to an image registry. Image registry can be local or public such as Docker Hub. Nuclio and Kubeless uses the same procedure while building and handling container images. Fission instead does not need local Docker installation or repositories. It is based on the notion of environments where functions are executed. The environments are pools of containers with language runtimes. Fission keeps a set of images in memory containing the runtimes, run them in containers and injects functions code into the running container. OpenWhisk is a another serverless framework that maintains a pool of containers with language runtimes. AWS Lambda runs functions similar way with Fission and OpenWhisk.

To summarize, Kubeless and Fission have an architecture that makes use of Kubernetes natively as much as possible, albeit all frameworks can support Kubernetes. OpenFaaS is flexible and highly extendable to support multiple container orchestrators and different architectures. The drawback of Nuclio's open-source version is that it does not support autoscaling. OpenFaas and AWS Lambda (through AWS Greengrass) are the only platforms supporting ARM, but on the other hand, neither of them supports GPU.

Feature	Kubeless	OpenWhisk	Fission	OpenFaaS	Nuclio	AWS Lambda
Open Source	Yes	Yes	Yes	Yes	Yes	No
License	Apache 2.0	Apache 2.0	Apache 2.0	MIT	Apache 2.0	
Commercial version	No	Yes	No	No	Yes	Yes
ARM support	No	No	No	Yes	No	via AWS Greengrass
GPU support	No	No	No	No	Yes	No
Supported programming languages	Python, Node.js, Ruby, PHP, Golang, .NET, Ballerina and custom	NodeJS, Swift, Java, Go, Scala, Python, PHP, Ruby, Ballerina and custom	Python, NodeJS, Go, C#, PHP and custom	C#, Dockerfile, Dockerfile for ARMHF, Go, Java, NodeJS, PHP, Python, Ruby	Go, Python, Shell, NodeJs, Java, .NET Core 2.0	Java, Go, PowerShell, Node.js, C Sharp, Python, Ruby
Container orchestrator support	Kubernetes native	Not required, Kubernetes support	Kubernetes native	Kubernetes, Docker Swarm, Nomad	Kubernetes	None
Function Triggers	HTTP, event, schedule	HTTP, event, schedule	HTTP, event, schedule	HTTP, event	HTTP, cron, Azure Event Hubs Trigger, v3io Stream	Lambda console, the Lambda API, the AWS SDK, the AWS CLI, AWS toolkits
Message queue integration	Kafka, NATS	Kafka	NATS, Azure storage queue	NATS, Kafka	Kafka, NATS, Kinesis, RabbitMQ, MQTT	Amazon Simple Queue Service (Amazon SQS)
CLI support	Yes	Yes	Yes	Yes	Yes	Yes
Autoscaling	Yes	Yes	Yes	Yes	In commercial version	Yes

Table 2: Comparison of features in serverless frameworks.

4 Related work

Serverless computing has drawn significant attention[5] in research [40][41][42] and open-source community. Emerging trends in academic research have been evaluate performance, security and machine learning solutions capabilities of FaaS offerings. Open source Serverless Frameworks are under of continuous development and research[1]. Studies have shown that first generation of open source serverless frameworks are still immature for commercial use [43]. In this chapter we exhibit most recent research trends of Serverless Computing and discussing where the current open source FaaS frameworks are heading.

Recent studies [3][6] have shown that serverless platforms and runtimes could be use in machine learning model hyperparameter tuning and serving deep learning models. Even some finding have been promising, well known serverless framework limitation "cold start" has been the main obstacle utilizing machine learning with serverless computing. Serverless frameworks notable latency problem caused by "cold start" combine with machine learning applications seem to be main bottleneck in current studies.

Since AI and machine learning are one of the hot topics in technology[44] and research, it is left to be seen how these trends affect into development of serverless frameworks. Open source development aim to make serverless frameworks more suitable for machine learning, But implementations are still in the early stage. Some frameworks such as Nuclio is already stretching the paradigm for making functions less "stateless" by allowing them to access local file systems and databases to safe their own state. Most of the serverless platforms like Fission, Kubeless and OpenFaas are allocating idle resources for mitigating effect of "cold start" and new worker allocation algorithm proposals[45] emerging to make serverless frameworks more warm for function invocations.

Research papers have proposed alternative way of running serverless functions, both in hardware level and different virtualization environments. Unikernel solutions have been in the scope of general microservice application studies [46] [26] and Ericsson Research studies shown unikernels to be a good virtualization technique for Faas [47]. Kim et al.[4] propose to run serverless workloads on graphics processing units with NVIDIA-docker and IronFunctions framework and evaluation shows that GPU outperformed processing of the CPU. Trustworthiness, cryptography and serverless security is one of the fast growing research areas. Even the amount of such studies is still limited, some novel approaches like running FaaS in Intel Software Guard Extensions (Intel SGX)[48], have been proposed by researchers. Se-lambda[49] is proposed to solve trustworthy issues by using SGX enclave to run sandboxed environment to prevent malicious cloud users and providers.

Current technology trends have and will be affecting the research of serverless computing. In reference to recent article *Nine Communications Technology Trends for 2019*[44], it is obvious that research trends are following technology, while machine learning, 5G and LTE evolution, MIMO and security leading the way for serverless

framework development. Serverless has not yet been proven to be solid option to build complicated distributed application and services. It remains to be seen what future holds for serverless computing both in research and industry applications.

5 Design and Implementation

AWS IoT Greengrass [27] is Google's AWS extension for edge devices. AWS IoT Greengrass utilizes AWS Lambda [28] functions to calculate predictions with machine learning models at the edge. However, the models are created, trained, and optimized in the central cloud. IoT Greengrass gives the developer the ability to use pre-trained models from Amazon SageMaker [29] or Amazon S3 [30].

In our experiments, we are using Raspberry Pi as the edge platform. We implement some characteristics of the AWS Lambda by utilizing Docker, Kubernetes, and OpenFaas framework. We benchmark OpenFaas function based solution against Kubernetes Service solution using simple Python3 based scripts that input three static parameters and output prediction values based on the parameters. Our purpose is not to evaluate the efficiency of the machine learning algorithm itself, but rather to use it as a tool compare the capabilities of serverless functions to their Kubernetes Service counterparts. We also benchmark OpenFaas autoscaling functionalities that allows a function to scale up depending on demand.

The Kubernetes testbed runs a simple Python http.server as a Kubernetes Service to serve incoming requests. The serverless testbed contains the same program code and runs on top of Kubernetes with OpenFaas watchdog working as a proxy for container. Our purpose is to evaluate which of the approaches provides lower latency and better quality of service in the case of concurrent requests.

5.1 Hardware setup

Our cluster hardware architecture consists of two Raspberry Pis, i.e., single-board computers. Raspberry Pi B 3+ equipped with a 32 Gb micro SD card serves as the master node (K8S-master) and Raspberry Pi 3 B equipped with a 16 Gb micro SD card acts as a cluster worker (K8S-worker-01). We have installed Raspbian Stretch Lite as the host operating system running on Linux kernel version 4.14. The nodes in the cluster are interconnected using a commodity Gigabit Ethernet switch. We utilize Weave (2.5.1) network plugin to implement pod networking in the Kubernetes cluster. The nodes obtain the IP addresses from an external DHCP server.

[27] https://aws.amazon.com/greengrass/
[28] https://aws.amazon.com/lambda/
[29] https://aws.amazon.com/sagemaker/
[30] https://aws.amazon.com/s3/

Spec	Raspberry Pi 3 B	Raspberry Pi 3 B+
CPU type/speed	ARM Cortex-A53 1.2GHz	ARM Cortex-A53 1.4GHz
RAM size	1GB SRAM	1GB SRAM
Integrated Wi-Fi	2.4GHz	2.4GHz and 5GHz
Ethernet speed	10/100 Mbps	300Mbps
PoE	No	Yes
Bluetooth	4.1	4.2

5.2 Kubernetes bootstrap

We utilize Kubeadm toolkit in order to bootstrap the Kubernetes (v1.14.1) cluster. We initialize the master node with "kubeadm init" command. We have disabled the swap partition to prevent early SD card failures. This initialization creates the needed cluster resources and generates a secure token for the cluster worker node to connect to cluster with "kubeadm join" command. We used Docker version 18.09.0 with API version 1.39.

5.3 OpenFaaS installation

Serverless framework offerings for ARM-based computers are still rare and OpenFaas was the only one that we have been able to run on Raspberry Pi. Our testbed utilizes ARM faas-netes installation which enables OpenFaaS usage on top of Kubernetes. Our OpenFaaS installation contains the following components:

- Alert manager (v 0.15.0-rc.0-armhf)
- Gateway (v 0.11.0-armhf)
- Gaas-netes (v 0.6.3-armhf)
- Prometheus (v 2.2.0-armhf)
- Queue worker (v 0.4.9-armhf)

We also install faas-cli (v0.8.3) tool that allows functions to be created and pushed to a local Docker repository.

5.4 Benchmarking tools

We used Hey[31] performance testing tool to send HTTP requests to OpenFaas and Kubernetes Service based prediction service. We used go version go1.11.1 linux/arm to run Hey on our cluster masternode. Dstat (0.7.2) was used to monitor CPU usage during OpenFaas autoscaling stress tests and Grafana (v6.1.6) was used to obtain OpenFaas pod scaling metrics, throughput and request responses.

[31]https://github.com/rakyll/hey

6 Evaluation

In this section, we evaluate performance, throughput and error tolerance of our serverless and Kubernetes Service respectively. The main scope of this evaluation is to clarify how much overhead serverless frameworks incur in systems with limited computing resources, such as Raspberry Pi. We test OpenFaas autoscaling capabilities to find out how many functions our test bed can run without affecting the performance.

6.1 OpenFaas autoscaling performance

First, we measure the impact of autoscaling in our OpenFaas set up. Our goal is to find a threshold where the replica count begins to affect the performance. OpenFaas is running a single replica for each deployed function by default which reduces latency and mitigates "cold start" during function invocation. We employ the OpenFaas default configuration that enables autoscaling starting from a single replica, scaling up to a maximum of 20 replicas.

We stress test a function with 2000 requests using a simple python3 script that invokes the function via HTTP by sending the parameters as json and then receives a resulting prediction value from the function. We run our prediction script as a serverless function in HTTP watchdog mode because it will most likely become the default mode for future OpenFaaS templates, and it is optimized to lower the latency and to support persistent connections by keeping the processes implementing the functions warm. We observe the CPU proportional usage of the cluster worker node with Dstat, and monitor pod scaling using Grafana. We run the test five times, and collect data from Grafana and Dstat in csv format, and merge pod metrics with the CPU usage data based on timestamps.

In our OpenFaas autoscaling tests, we utilize the default Alert Manager implementation of the framework. OpenFaas also enables use of Kubernetes Horizontal Pod Autoscaler (HPA)[32] as an alternative.

Unfortunately, we were not able to test any HPA based autoscaling features in our test bed, so we cannot test Kubernetes Service autoscaling performance at all. HPA uses a Metrics Server[33] to collects metrics from the Kubernetes, but Metric Server has no support for ARM based devices.

We notice that the response time is roughly the same when the system operates under nine redundant function pods. The median response time within one to nine function replicas is 56.0 milliseconds, with the slowest and fastest response time differing only 1.7 milliseconds. Consequently, serving simple functions like this indicates that it is possible to serve a high load of requests with just a single replica without impacting unfavourably the performance. When the function scale over ten

[32]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[33]https://github.com/kubernetes-sigs/metrics-server

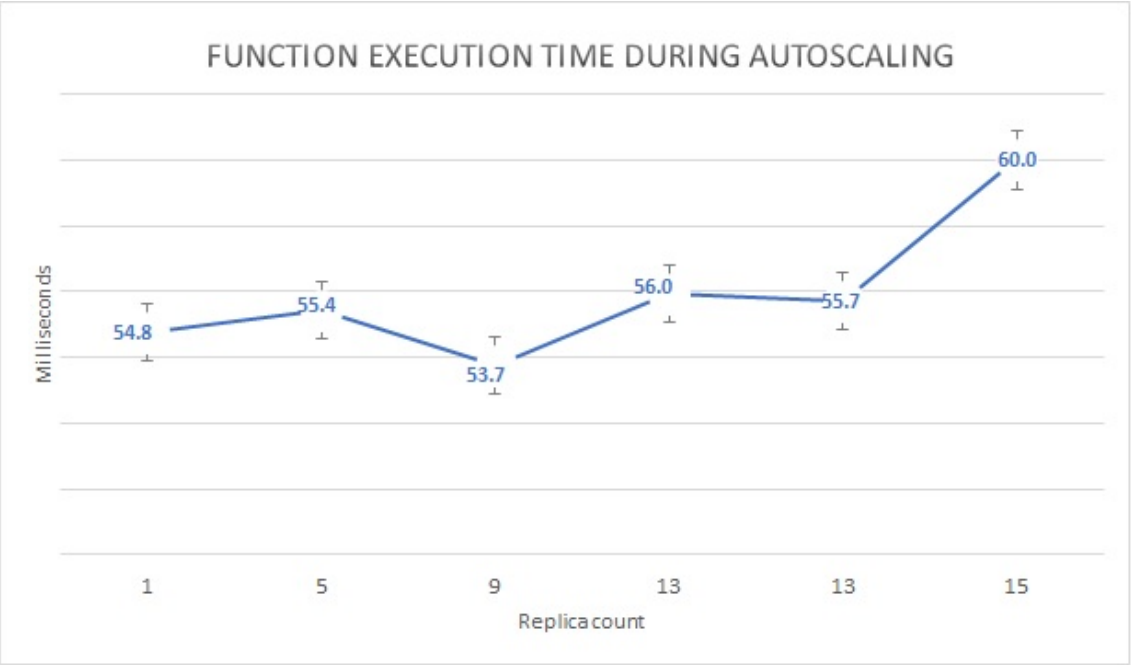

Figure 14: OpenFaas execution time during autoscaling.

replicas, its performance start to diminish.(Figure 14)

Figure 15 shows the CPU usage % of user, system, idle and waiting processes during autoscaling. We notice a significant increase of waiting processes when OpenFaas autoscaling operates between 9 and 13 pods and starts to prepare new pods to scale. We notice that system processes consumed increasingly more CPU resources when the amount of replicas increased to thirteen pods.

In three out of our five test runs OpenFaas crashed after replica count reached thirteen pods. We were able to collect some data from the system while the pod count was fifteen pods. After system failed, it took several minutes for OpenFaas to resume running state. In some cases, we had to reboot the cluster nodes in order to revive the system.

A detailed examination of the results in figure 14 reveals that the response time is at its best somewhere between five and nine replicas. But as mentioned before, the amount of idle processes increases while replica count increase and operating system reserve resources to scale new replicas. So, in this application, restricting function autoscaling to maximum seven replicas during the deployment would result in the best performance. It is worth noting that OpenFaas allows to set a policy of minimum and maximum amount of replicas for the function during function deployment with *com.openfaas.scale.min* and *com.openfaas.scale.max* labels. We use these labels to set amount of warm function pods in the next section where we continue to measure how the quantity of concurrent users impacts OpenFaas.

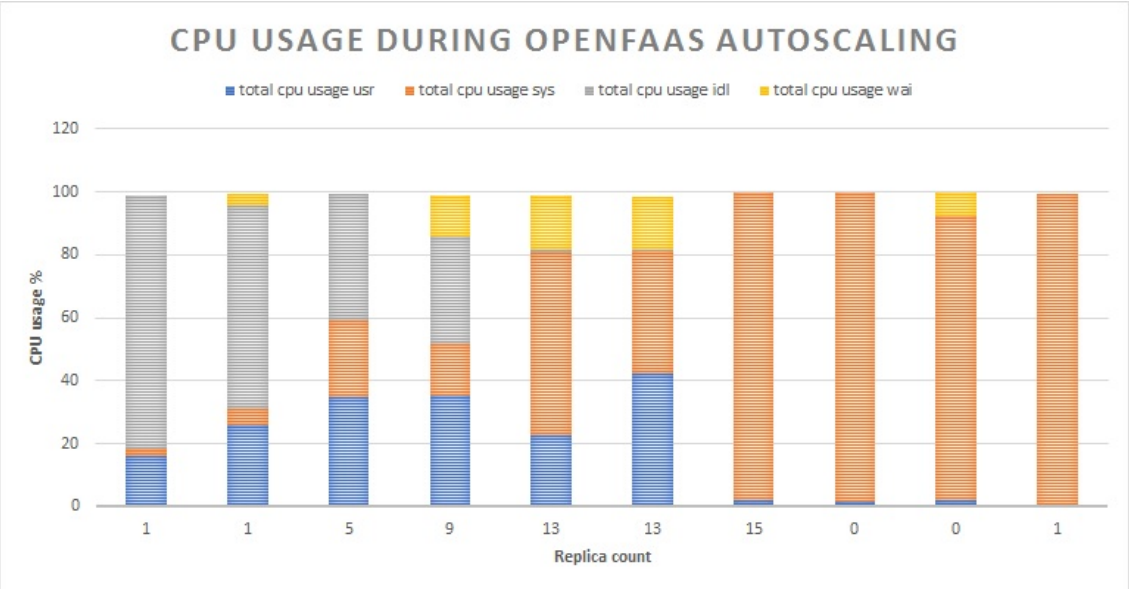

Figure 15: OpenFaas CPU usage during autoscaling.

6.2 Impact of concurrent users

We use Hey [34] performance testing tool to monitor system throughput and server response. We send a total of 1000 HTTP requests either to OpenFaas or Kubernetes Service testbed, and vary the amount of concurrent requests (1, 5, 10, 20, 50, 100) during different test runs. We repeat each experiment five times and calculate the average result. We disable auto scaling, i.e., run a fixed number of replicas in both set ups. This way, we avoid any increase in the response times that might occur when using auto scaling. We test with different amount of function/containers: 1, 5 and 7 replicas. For OpenFaas deployment, we use *com.openfaas.scale.min* and *com.openfaas.scale.max* labels and set them to equal value, depending on the test case. Similarly, we scale our deployment with *kubectl scale –replicas=x* command in the Kubernetes Service version.

With OpenFaas as shown in figure 16, we note that when the number of concurrent requests is low (1, 5, 10), the response times do not show significant difference with varying amount of replicas. However, when the amount of concurrent requests rises to 20, the serverless implementation starts to show better performance with higher replica counts. In fact, OpenFaas shows an average decrease of 27.5 % in the response time when function replica count approximates 5, and concurrent request volume is between 20 and 100.

The Kubernetes testbed in figure 17 shows a similar trend as the OpenFaaS one, but a similar phenomenon occurs with lower concurrency levels as well. In all experiments, the Kubernetes Service shows better response times than OpenFaas. This indicates that the OpenFaaS framework itself brings significant overhead, and the

[34] https://github.com/rakyll/hey

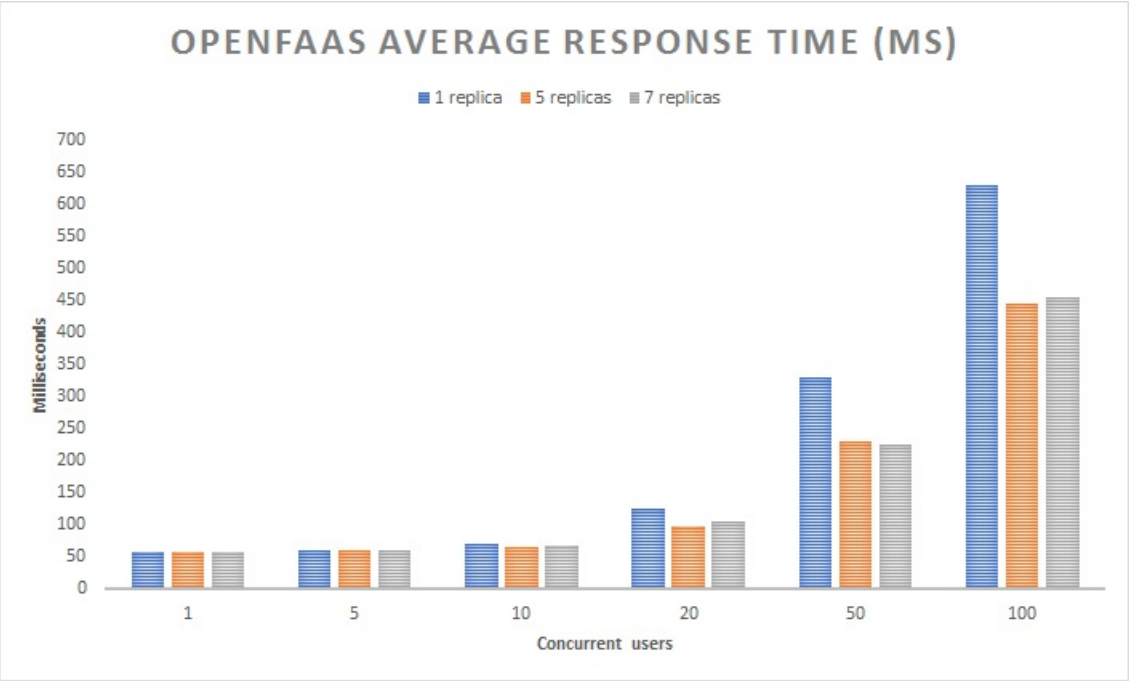

Figure 16: OpenFaas average response times with different replica and concurrent user counts.

serverless approach may not be the best choice for latency sensitive applications. For instance, comparing the measurements result with 1 pod and 1 concurrent request, the average response time with serverless is 57,99 milliseconds and 5 milliseconds with Kubernetes Service. Also in the other end of the measurement range, i.e., with 7 replicas and 100 concurrent requests, the response time differs notably: OpenFaas responds in 454.68 milliseconds and Kubernetes in 164.28 milliseconds.

Next, we benchmark the sum of processing time of 1000 requests with varying levels of concurrency. Figure 18 shows (with y-axis in log scale) how different replica counts and concurrent request levels affect the total calculation times. A closer examination of the results shows that Kubernetes Service outperforms OpenFaaS in all of the measurements. However, with more concurrent requests, the execution time of Kubernetes Service descends first but then starts to ascend slowly. With higher levels of concurrency, OpenFaaS seems to converge towards Kubernetes Service performance. A possible explanation for this is that OpenFaaS introduces a fixed performance overhead that is visible with smaller workloads but larger workloads mask this fixed overhead.

Lastly, we evaluate how many request per second the two testbeds can handle in different replica and concurrency levels. Figure 19 shows our findings. A closer look at the results shows that Kubernetes Service can handle more requests per second in most of the cases. The only notable Kubernetes performance drop occurs when it is serving many concurrent requests with a single pod. In the best case (with 7 replicas, 20 concurrent requests), Kubernetes is able to handle an average of 533.2

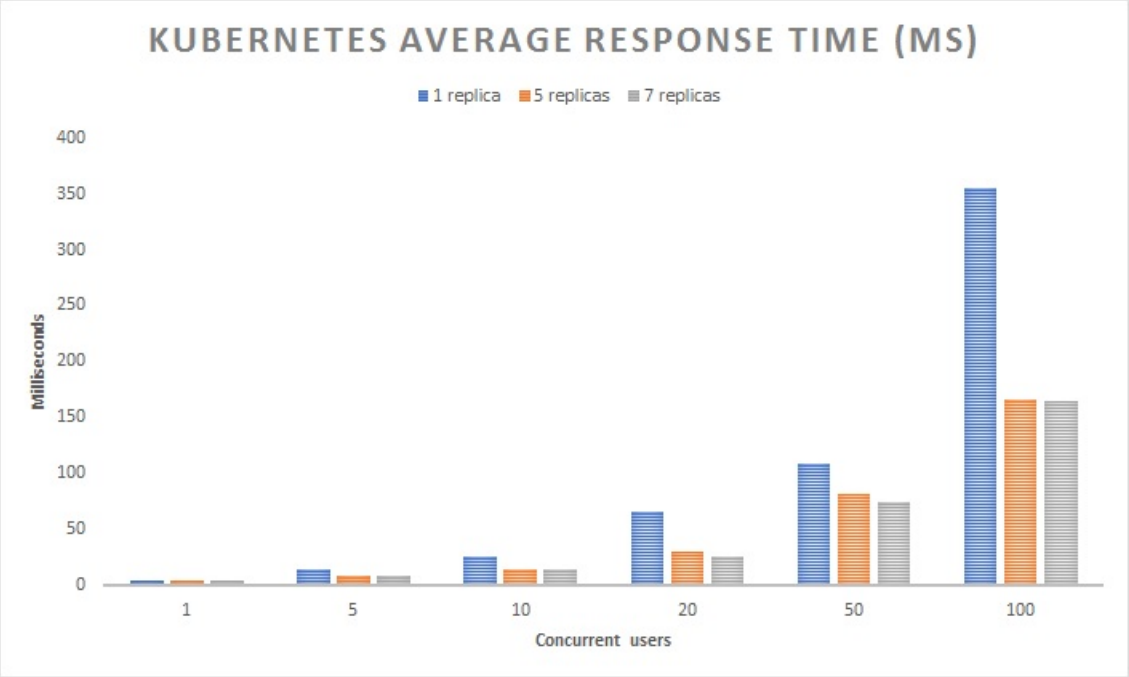

Figure 17: Kubernetes average response times with different replica and concurrent user counts.

requests per second. In the same measurement, OpenFaas can process an average 176.2 requests per second.

In test runs related to figure 19, OpenFaas did not fail in any of test runs but Kubernetes Service failed when we deployed only a single pod. We collected the success ratio of all experiments, and marked responses without 200 response code as a failure. We observed that OpenFaas success ratio was 100 %, and that all HTTP responses were received successfully with the 200 response code. However, we noticed that the success ratio of Kubernetes dropped to 99,98 % because the framework failed to respond several times when the replica count was 1 and concurrency level was 100.

7 Discussion and Future Work

Our test results show that the Kubernetes Service testbed has faster response time across all test cases when compared to OpenFaaS. In Kubernetes, each HTTP request is processed by the API and the Service that balances load to worker pods. In OpenFaas, each request and response is processed by the Openfaas API *gateway*, *faas-netes* and *watchdog*. Hence, OpenFaas framework adds overhead, and, while its design may change in the future, it seems that OpenFaaS is not yet mature enough to serve latency sensitive applications on edge devices based on our experiments.

Our experiments shows that CPU-intensive tasks, such as autoscaling, need to be

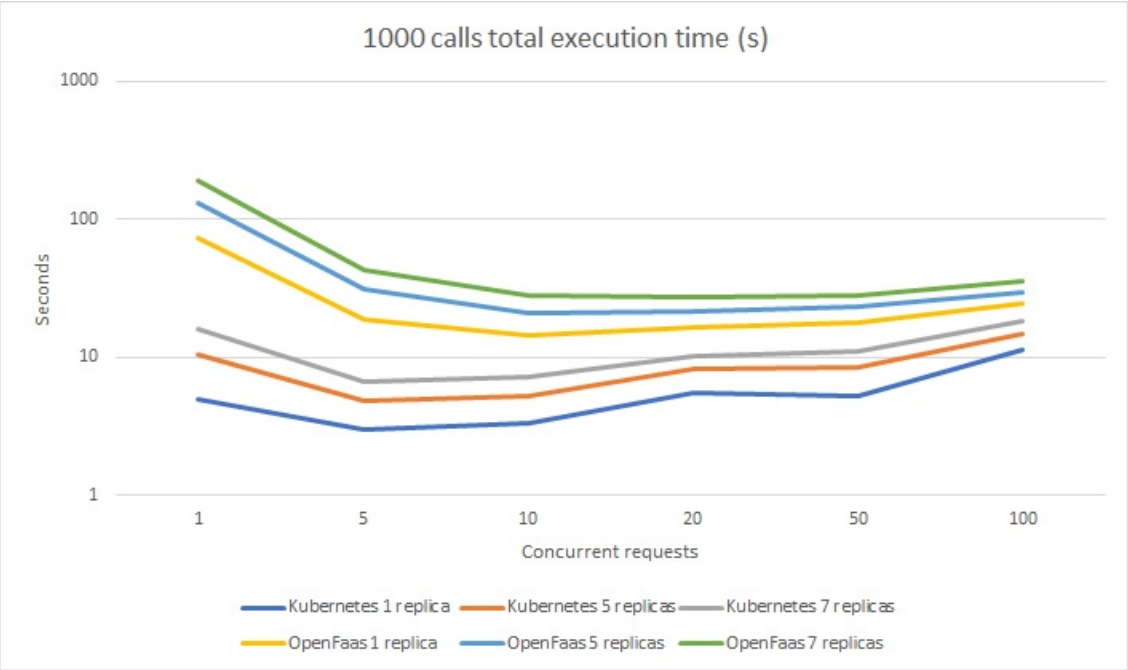

Figure 18: OpenFaas and Kubernetes processing time for 1000 request with varying replica counts and varying amount of concurrent requests.

limited on a constrained devices such as Raspberry Pi. Careless autoscaling can cause severe problems and need to be bound to a maximum replica value. Our test bed shows that running seven pod replicas caused problems for the Kubernetes even when autoscaling was disabled. When replica count exceeds nine replicas, the system becomes unresponsive. It is unfortunate that we were unable to experiment Kubernetes Horizontal Pod Autoscaler due to installation problems with metrics server[35], and this remains future work for us.

Our study raises another question. How a serverless framework such as OpenFaas can save resources of a device if it keeps some framework resources such as pods warm for function invocation? Scaling replicas to zero is actually possible in OpenFaaS using faas-idler [36] component but it does not yet have ARM support. So we were not able to use faas-idler to test OpenFaas "Cold Start" performance on ARM, and this remains future work.

Initially, our plan was to test both of the testbeds with 10 warm replicas, but both testbeds became unresponsive regardless of which implementation we used. Our Raspberry Pi cluster was unable to handle the workload of running 10 replicas. This indicates that constrained devices are not able to run multiple replicas at the same time, even if the service itself is simple as our implementation of prediction edge service. We lower the maximum replica count to 7 to save time.

OpenFaas adds additional performance overhead on top of the Kubernetes. Our

[35]https://github.com/kubernetes-incubator/metrics-server
[36]https://github.com/openfaas-incubator/faas-idler

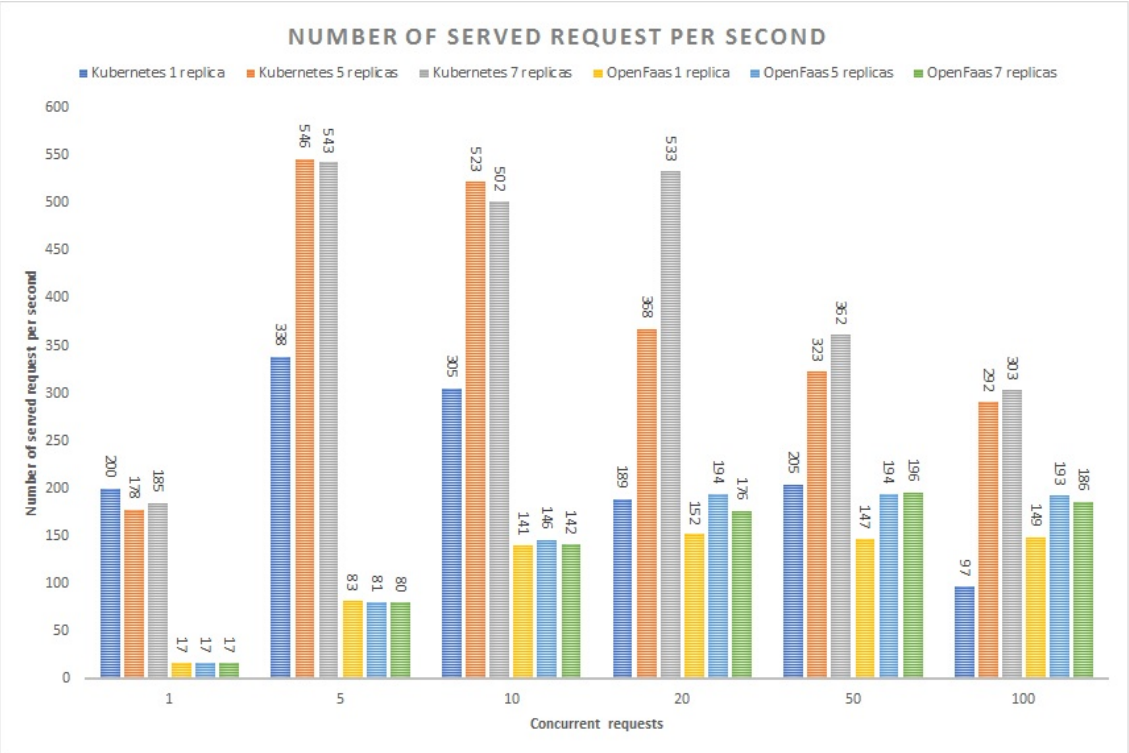

Figure 19: Number of requests that Openfaas and Kubernetes can serve per second.

tests raised the question of whether a single prediction service could be fully deployed without Kubernetes and Openfaas. On constrained devices, small services could just be run with Docker containers without Kubernetes. We assume that Docker can serve a moderate amount of users without notable latency and can be comparable of running a Kubernetes Service with single pod replica. To define how much additional overhead Kubernetes itself bring to response times remains future work. Running a simple asynchronous web server without any virtualization techniques on a Raspberry Pi could be a better option.

If services cannot efficiently be executed on thin edge devices at the moment, then what kind of devices should then be used at the edge in the future? Single board computers are adequate at least for collecting data. Our experiments indicate that a vanilla Raspberry Pi is not necessary the most ideal device to be used for machine learning models. In the future, edge and IOT devices can be equipped with more computing power and resources. Today's single board computers are convenient for prototyping but not yet mature for latency critical services in production use. For example, 5G promises lower than 1 ms response times and such latency cannot not be achieved with the current Raspberry Pi versions. It is promising that our Kubernetes Service testbed was achieving a 5 millisecond response time with no replica and no concurrency [37].

[37]It is worth noting that our experiments were run in 10/100 Mbit ethernet, and the use of wireless technologies would add extra latency

8 Conclusions

Serverless computing is a promising paradigm and its commercial implementations are widely in use. In theory, Serverless and Function as a Service can provide distributed edge cloud based on open-source serverless frameworks and Kubernetes. This can notably speed up the development of the edge, and the use of open-source can lower the development cost. On the other hand, open-source serverless frameworks are still quite immature for reliable edge cloud usage, and we believe further development of serverless frameworks is still needed.

Cloud service providers usually have high degree of vendor lock-in in their serverless offerings. Development of open-source serverless frameworks is also needed to mitigate the high degree of vendor lock-in.

With open-source frameworks, developers can choose a framework with the desired runtimes that suite their needs in application development. In contrast, in commercial cloud service models, the vendor is in total control of the underlying infrastructure and the developers need to adapt to specific programming languages and design patterns to deploy their serverless applications. In general, most of the commercial FaaS offerings support common programming languages like Python, NodeJs and Golang. On the other hand, some open-source serverless frameworks, such as Fission and OpenFaaS, allow developers to extend the programming language support in the framework.

In this thesis, we show a qualitative comparison between five open-source serverless frameworks. These frameworks are not compared to any commercial serverless offerings such as AWS Lambda. We depict the features and architectural choices of different open source frameworks. Our study shows that many open-source framework do not always fulfill all the requirements of serverless computing. For example, it is quite common that these frameworks keep some resources idle and some frameworks even tend to stretch the principle of stateless functions itself. One example is Azure Functions extension called Durable Functions that lets developers write stateful functions in a serverless compute environment.

We also show our benchmarking results with OpenFaaS framework running on an Kubernetes edge cloud (i.e., a Raspberry Pi) based on algorithms typically utilized in machine learning. Our test results and comparison show that serverless frameworks such as OpenFaas may add considerable overhead to small edge clouds. In our test cases, we notice that careless autoscaling of serverless function may cause severe problems on constrained edge devices. Even though our test environment is very limited, our findings support the understanding that further development of open-source serverless frameworks is needed to tackle complex problems at the edge of the network.

References

1 S. K. Mohanty, G. Premsankar, and M. di Francesco, "An evaluation of open source serverless computing frameworks," in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 115–120, Dec 2018.

2 S. K. Mohanty, "Evaluation of serverless computing frame-works based on kubernetes," Master's thesis, Aalto University, School of Science, Degree Programme of Computer Science and Engineering, 2018.

3 V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 257–262, April 2018.

4 J. Kim, T. J. Jun, D. Kang, D. Kim, and D. Kim, "Gpu enabled serverless computing framework," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pp. 533–540, March 2018.

5 I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, *Serverless Computing: Current Trends and Open Problems*, pp. 1–20. Singapore: Springer Singapore, 2017.

6 L. Feng, P. Kudva, D. Da Silva, and J. Hu, "Exploring serverless computing for neural network training," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 334–341, July 2018.

7 B. P. Rimal, A. Jukan, D. Katsaros, and Y. Goeleven, "Architectural requirements for cloud computing systems: an enterprise cloud approach," *The Computer Journal*, vol. 9, no. 1, p. 3–26, 2011.

8 E. van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann, "A spec rg cloud group's vision on the performance challenges of faas cloud architectures," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, (New York, NY, USA), pp. 21–24, ACM, 2018.

9 A. Eivy, "Be wary of the economics of "serverless" cloud computing," *IEEE Cloud Computing*, vol. 4, pp. 6–12, March 2017.

10 H. Atlam, R. Walters, and G. Wills, "Fog computing and the internet of things: A review," *Big Data and Cognitive Computing*, vol. 2, 04 2018.

11 K. Bierzynski, A. Escobar, and M. Eberl, "Cloud, fog and edge: Cooperation for the future?," in *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pp. 62–67, May 2017.

12 Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing—a key technology towards 5g," *White Paper*, p. 1–36, 2014.

13 T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration," *IEEE Communications Surveys Tutorials*, vol. 19, pp. 1657–1681, thirdquarter 2017.

14 N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet of Things Journal*, vol. 5, pp. 450–465, Feb 2018.

15 Y. Yu, "Mobile edge computing towards 5g: Vision, recent progress, and open challenges," *China Communications*, vol. 13, pp. 89–99, N 2016.

16 Z. Zhang and S. Li, "A survey of computational offloading in mobile cloud computing," in *2016 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pp. 81–82, March 2016.

17 E. Calvanese Strinati, S. Barbarossa, J. L. Gonzalez-Jimenez, D. Ktenas, N. Cassiau, L. Maret, and C. Dehos, "6g: The next frontier: From holographic messaging to artificial intelligence using subterahertz and visible light communication," *IEEE Vehicular Technology Magazine*, vol. 14, pp. 42–50, Sep. 2019.

18 S. Hung, H. Hsu, S. Lien, and K. Chen, "Architecture harmonization between cloud radio access networks and fog networks," *IEEE Access*, vol. 3, pp. 3019–3034, 2015.

19 D. C. Marinescu, *Cloud Computing : Theory and Practice.*, vol. 1st ed. Morgan Kaufmann, 2013.

20 T. Takahashi, G. Blanc, Y. Kadobayashi, D. Fall, H. Hazeyama, and S. Matsuo, "Enabling secure multitenancy in cloud computing: Challenges and approaches," in *2012 2nd Baltic Congress on Future Internet Communications*, pp. 72–79, April 2012.

21 W. J. Brown, V. Anderson, and Q. Tan, "Multitenancy - security risks and countermeasures," in *2012 15th International Conference on Network-Based Information Systems*, pp. 7–13, Sep. 2012.

22 S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 275–287, Mar. 2007.

23 B. Xavier, T. Ferreto, and L. Jersak, "Time provisioning evaluation of kvm, docker and unikernels in a cloud platform," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 277–280, May 2016.

24 V. Cozzolino, A. Y. Ding, and J. Ott, "Fades: Fine-grained edge offloading with unikernels," in *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, HotConNet '17, (New York, NY, USA), pp. 36–41, ACM, 2017.

25 R. Pavlicek, "Unikernels."

26 T. Goethals, M. Sebrechts, A. Atrey, B. Volckaert, and F. De Turck, "Unikernels vs containers: An in-depth benchmarking study in the context of microservice applications," in *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pp. 1–8, Nov 2018.

27 M. Kapuruge, J. Han, and A. Colman, "Bibliography," in *Service Orchestration As Organization*, pp. 257 – 272, Boston: Elsevier, 2014.

28 E. Wolff, *Microservices: Flexible Software Architectures*. CreateSpace Independent Publishing Platform, 2016.

29 S. Makam, *Mastering CoreOS.* Community Experience Distilled, Packt Publishing, 2016.

30 D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, (Berkeley, CA, USA), pp. 305–320, USENIX Association, 2014.

31 M. H. Miraz, M. Ali, P. S. Excell, and R. Picking, "A review on internet of things (iot), internet of everything (ioe) and internet of nano things (iont)," *IEEE Conferences*, pp. 219–224, 09 2015. Date revised - 2016-04-01; Last updated - 2016-04-04.

32 S. Agrawal, A. Bansal, and S. Rathor, "Prediction of sgemm gpu kernel performance using supervised and unsupervised machine learning techniques," in *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pp. 1–7, July 2018.

33 H. Ando, Y. Niitsu, M. Hirasawa, H. Teduka, and M. Yajima, "Improvements of classification accuracy of film defects by using gpu-accelerated image processing and machine learning frameworks," in *2016 Nicograph International (NicoInt)*, pp. 83–87, July 2016.

34 M. A. H. B. Sulaiman, A. Suliman, and A. R. Ahmad, "Measuring gpu-accelerated parallel svm performance using large datasets for multi-class machine learning problem," in *Proceedings of the 6th International Conference on Information Technology and Multimedia*, pp. 299–302, Nov 2014.

35 J. Lew, D. A. Shah, S. Pati, S. Cattell, M. Zhang, A. Sandhupatla, C. Ng, N. Goli, M. D. Sinclair, T. G. Rogers, and T. M. Aamodt, "Analyzing machine learning

workloads using a detailed gpu simulator," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 151–152, March 2019.

36 M. Hausenblas, *Serverless Ops - A Beginner's Guide to AWS Lambda and Beyond*. O'Reilly Media, Inc, 2017.

37 D. Géhberger and A. Prekopcsák, "Er-tip technology assessment: Analysis of open source function-as-a-service frameworks," *Report*, 2018.

38 M. Sewak and S. Singh, "Winning in the era of serverless computing and function as a service," in *2018 3rd International Conference for Convergence in Technology (I2CT)*, pp. 1–5, April 2018.

39 Y. Haviv, "nuclio: The new serverless superhero," 2017. Last accessed 2 May 2019.

40 L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, (Boston, MA), pp. 133–146, USENIX Association, 2018.

41 E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, (New York, NY, USA), pp. 445–451, ACM, 2017.

42 A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi, "Understanding ephemeral storage for serverless analytics," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, (Boston, MA), pp. 789–794, USENIX Association, 2018.

43 J. Hellerstein, J. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," 12 2018.

44 Y. Qu, A. Lozano, and A. Gatherer, "Nine communications technology trends for 2019," 2019. Last accessed 9 May 2019.

45 Y. Kim and G. Cha, "Design of the cost effective execution worker scheduling algorithm for faas platform using two-step allocation and dynamic scaling," in *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pp. 131–134, Nov 2018.

46 B. Xavier, T. Ferreto, and L. Jersak, "Time provisioning evaluation of kvm, docker and unikernels in a cloud platform," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 277–280, May 2016.

47 D. Géhberger and D. Kovács, "Cooling down faas: Towards getting rid of warm starts," tech. rep., Ericsson Review, April 2019.

48 F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner, "S-faas: Trustworthy and accountable function-as-a-service using intel sgx," *CoRR*, vol. abs/1810.06080, 2018.

49 W. Qiang, Z. Dong, and H. Jin, "Se-lambda: Securing privacy-sensitive serverless applications using sgx enclave," in *Security and Privacy in Communication Networks* (R. Beyah, B. Chang, Y. Li, and S. Zhu, eds.), (Cham), pp. 451–470, Springer International Publishing, 2018.

50 M. Westerlund and N. Kratzke, "Towards distributed clouds: A review about the evolution of centralized cloud computing, distributed ledger technologies, and a foresight on unifying opportunities and security implications," in *2018 International Conference on High Performance Computing Simulation (HPCS)*, pp. 655–663, July 2018.

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

52 B. Xavier, T. Ferreto, and L. Jersak, "Time provisioning evaluation of kvm, docker and unikernels in a cloud platform," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 277–280, May 2016.