FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Serverless Edge Analytics

Untersuchung über die derzeitigen Stärken und Schwächen von Serverless Edge Analytics unter Berücksichtigung derzeitiger Werkzeuge und Frameworks

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Business Informatics

eingereicht von

Michael Pazourek, BSc
Matrikelnummer (0)1225721

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar

Wien, 30. November 2019

Michael Pazourek	Schahram Dustdar

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Serverless Edge Analytics

Investigation on the strengths and drawbacks of Serverless Edge Analytics with current tools and frameworks

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Michael Pazourek, BSc
Registration Number (0)1225721

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Schahram Dustdar

Vienna, 30th November, 2019

Michael Pazourek | Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Michael Pazourek, BSc

Vorgartenstraße 88/9
A-1200 Wien
Austria

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30. November 2019

Michael Pazourek

Acknowledgements

A special thank you belongs to my advisor Schahram Dustdar who gave me the opportunity to write this Diplomarbeit at his department. I want to thank him for his extraordinary inspiration, his generous patience and motivational support.

Furthermore, I want to thank my family for enabling and supporting my education. Moreover, my girlfriend and all friends who had an open-ear for discussing the problems of this thesis have my undying gratitude.

Kurzfassung

Während der Ausbau von Internetinfrastrukur in den letzten Jahren ins stocken geriet, stieg die Menge an zu verarbeitenden Daten dramatisch. Dadurch entstand die Situation, dass insbesonders jene Daten, welche auf Endgeräten generiert werden, heute großteils nicht für Datenanalysen genutzt werden können. State of the Art Machine Learning benötigt enorme Mengen an Trainingsdaten um Vorhersagemodelle erstellen zu können und würde von all jenen Daten profitieren, welche derzeit nur am Rande des Internets, auf Endgeräten, verfügbar sind. Edge Computing verspricht, die Latenzprobleme der derzeitigen Internetinfrastuktur zu reduzieren. Serverless Computing ist zudem eine Neuerung welche vorangig im Cloud Computing entstanden ist, aber auch auf Edge Geräten sinnvoll erscheint. Iin letzter Zeit entstehen einige Frameworks, welche Serverless Computing auch auf Edgegeräten ermöglichen.

Der Zweck dieser Diplomarbeit ist die Erforschung von Serverless Edge Analytics. Um dies zu erforschen, wurden zu Beginn der Arbeit umfassende Analysen über bestehende Frameworks, die Anforderungen durch Machine Learning, sowie die Anforderungen an Monitoring von Serverless Functions, basierend auf relevanter Literatur und vertrauenswürdigen Onlinequellen durchgeführt. Anschließend wurde die Umsetzbarkeit sowie die Sinnhaftigkeit von dynamischem Weiterleiten von Serverlessauslösern (Triggers), von Edge auf Cloud Geräte, am Beispiel von AWS Greengrass untersucht.

Die Erkenntnisse aus der Analyse und den Experimenten zeigen, dass Monitoring von AWS Greengrass derzeit nicht sinnvoll umgesetzt werden kann. Zudem haben die Experimente ergeben, dass das Forwarding von Serverless Triggern von AWS Greengrass an das cloudbasierte AWS Lambda, derzeit, mit der Standardimplementierung von AWS extrem hohe Antwortzeiten aufweist. Generell ergibt sich anhand der Ergebnisse dieser Diplomarbeit das Bild, dass aufgrund mehrerer Einschränkungen, Serverless Edge Computing mit AWS Greengrass und AWS IoT auf derzeitiger Edge Computing Hardware noch stark verbessert werden sollte.

Nichts desto trotz bestätigen die Ergebnisse dieser Diplomarbeit abermals diverse andere Forschungsarbeiten darin, dass Edge Computing für manche Anwendungsfälle sehr geeignet erscheint, dass aber insbesonders Serverless Edge Computing bzw. generell Serverless Computing nicht für Fälle mit Anspruch an niedrige Antwortzeiten (Latenz) geeignet erscheinen und dass zudem noch einige Einschränkungen die Verwendung von Machine Learning erschweren.

Abstract

Due to increasing amount of information that gets available at client devices, the centralized layout of the internet begins to be outdated. By now, we have tremendous amounts of data that is only available at end-devices from where data can, due to connection limitations and missing capacity at cloud data centers, not be sent to central locations for analysis. Machine learning training requires enormous amounts of input data to find meaningful patterns and would benefit by the availability of the data that can currently not be sent to centralized clouds. To reduce the resource overhead of cloud services, the Function as a Service (FaaS) paradigm was recently proposed as a possible improvement for cloud data centers. Novel frameworks bring serverless functions on edge computing devices too and could dramatically improve analytical services.

The aim of this work is the analysis of serverless edge computing under the context of applying machine learning at those edge devices in order to use data that can not easily be sent to the cloud. In order to do this, beside analysis of existing frameworks and literature, it was a task to investigate on the feasibility of implementing an extension for AWS Greengrass that allows dynamic forwarding of serverless function triggers to edge or cloud devices based on the utilization each AWS Greengrass edge node.

The findings of this thesis reveal that monitoring the processing time and utilization of AWS Greengrass nodes is not meaningful with today's solutions. As it was necessary for this thesis to research various serverless function frameworks and their monitoring approaches, an extensive study about their capabilities is given. Beside finding that monitoring of AWS Greengrass delivers vacuous results, it was found that forwarding triggers of serverless functions from AWS Greengrass to the cloud-based AWS Lambda results in extremely slow response times. As it was necessary to test machine learning workload on AWS Lambda, various insights and limitations were found during this process. In general it was found that serverless edge computing, edge computing hardware and AWS IoT are currently limited in many ways.

Non the less, while serverless computing has proven to be limited for low-latency applications, various possible further research fields, like Federated Learning, have been identified as promising ways to tackle the challenges of analyzing data within serverless edge computing and edge computing in the future.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The "*Cisco Global Cloud Index estimates that nearly 85 ZB (Zettabyte) of data will be generated by all people, machines, and things by 2021, up from 22 ZB generated in 2016*"[Cis16]. Furthermore, they also expect estimated traffic handled by data centers to only be as low as 21 ZB by 2021 (7 ZB in 2016).

It is clear to see that there is a significant 64 ZB (300% of processed data) gap between data that is generated and not transmitted to a data centers and data that will be processed in cloud computing environments. Ciscos Global Cloud Index notes that the mentioned gap "*will be the space for edge computing*"[Cis16] and that it is clear, that those systems will also influence how machine learning will be practiced in the future.

Researchers of distributed data analysis and machine learning are focusing on cloud computing while data amounts are growing at the edge of the internet. Generally, current trends in cloud computing include edge computing, big data analytics, stream analysis, dynamic resource provisioning, energy efficiency, live migration, security and serverless computing [CSB17].

Connecting the mentioned research approaches with the fact that edge computing will be relevant for handling the data amounts of the future results in the emerging field of edge analytics where data is analyzed and used in geographical proximity to where it is created. Bringing computational intelligence closer to the data sources and running machine learning tasks on edge devices can facilitate many use cases generally referred to as edge analytics. Examples by M. Satyanarayanan include data intensive tasks like video surveillance that uses high bandwidth between all cameras and the data analysis components traditionally hosted at a data center; masking cloud outages while still being able to respond to local events with local emitters; reducing the needed network bandwidth

by proposing scalability via edge analytics and having privacy-policy enforcement by not sending critical data to cloud data centers or other countries [Sat17].

Not only Cisco sees high potential in machine learning for edge computing [Cis16]: quite recently, considerable attention has been paid to distributed machine learning. Federated learning is currently discussed in various papers partly associable with Google [BIK+17], [KMRR16], [MMRyA16] where they focus on on-device training but this could also be used in the context of edge analytics with extremely low powered end devices.

Clear cut: beside the mentioned advances in edge computing and machine learning, the paradigm of serverless computing (also called Function as a Service - FaaS) is changing the way cloud computing is used. Serverless computing enables deployment of code with a clear structure, a function, to a managed cloud environment and is said to refine the microservice paradigm. Functions for a given environment only need to support a specified entry point and can then be triggered at any server of the provider without the need of language-specific environment setups and extensive DEV-Ops. Runtimes like the Java Virtual Machine or NodeJs do not have to be bundled with the code, as it is currently done for most container-based microservices, as they are already available at the servers of the provider. This obviously reduces the complexity of the deployment process, improves the utilization of server hardware (by not running the runtime in every container once) and therefore, scalability.

Recent research by Glikson, Nastic, Dustdar [GND17] and others [NRS+17] mention that these opportunities could also be used on edge devices to reduce the complexity of such systems. Extending serverless computing to the edge helps decoupling applications from the actual devices they run on and also reduces the administrative overhead of redeploying code to the devices. AWS has entered this field with their AWS Greengrass product line [Ser17] that can execute AWS Lambda functions, that were previously already used in cloud environments, on edge devices too. In 2019, however, as found in this thesis and discussed in Chapter 5, the framework currently only provides limited monitoring capabilities.

To the authors' best knowledge, acquired from literature research, the combination of analytical processes (especially machine learning), serverless computing and edge computing is a relatively unexplored area. Serverless edge analytics has advantages such as reduction of network utilization, a reduction of latency, is better for privacy and improves the data throughput. There are, however, still not many systems available in practice that use this architectural approach and additional research is justified.

Bringing the aforementioned concepts together, results in serverless edge analytics. Research and building actual solutions could open up new fields for applications but are challenging tasks. As mentioned in section 1.2, it is not always possible to execute any analytic function at any time on a low powered edge device, such as a Raspberry Pi. Based on the results of the the research group around Baresi et. al [BMG17], Edge Devices can easily be computationally overloaded. Then, some of the mentioned advantages of edge computing do not apply anymore. Cloud environments can, in contrast to Edge

Devices, scale to the needs of the users. Edge Devices clearly lack this ability.

While training machine learning models on a serverless edge analytics platform is yet unexplored, it would clearly need a lot of computational power. This thesis is therefore focusing on the exciting task to explore ways of measuring and scheduling the execution of serverless functions between the cloud and edge devices on the basis of the current device utilization.

1.2 Problem Definition

Serverless edge devices will easily be overloaded and then they will not be able to process all triggers in reasonable time [BMG17]. In contrast to cloud computing, edge computing can not be seen as a central entity. Edge computing nodes are located close to users and edge computing applications have to deal with a high degree of distribution computing. That is why edge devices can usually not be *scaled up* while cloud data centers scale well and provide a lot of concurrent instances of serverless functions. Using edge devices for advanced workload like analytical tasks can easily result in overloaded edge nodes.

As evident in Chapters 2 and 3, there is rising interest from both research and industry fields to apply edge analytics. Current use cases are limited to custom analytic functions and machine learning approaches and only focus on machine learning inferencing, not on the training steps. Mehdi Mohammadi et al. share the common perception that *"due to the data thirstiness of deep learning models, data transfers to the cloud may become a bottleneck"* [MAFSG17]. Using custom analytical functions or inferencing at edge devices reduces the amount of data that is sent to data centers. Current centralized cloud-based ML-approaches will then miss out a lot of training data, which was transferred to data centers in the past.

L.Feng and colleagues state that training machine learning models is time, *"compute and memory intensive and tightly coupled"* [FKSH18] and serverless functions are typically stateless, limited in memory and limited in execution time [FKSH18]. MxNet and TensorFlow already implement distributed training with multiple GPUs, however, using the parallelism of serverless runtimes has not been explored much yet [FKSH18].

Works about machine learning on serverless architecture exist, however, the approaches mostly used inferencing, not training of machine learning models [FKSH18]. Still, even with inferencing, overloading can easily happen. This is also confirmed by the experiments undertaken in Chapter 5.

Another problem is the availability of various frameworks for serverless computing, of whom only some support edge computing (such as AWS Lambda, Microsoft Azure Functions). Further research will require a discussion on the limitations, features and general communication approaches of the available frameworks. It is important to identify how the communication interfaces could support the synchronization steps of federated learning processes. Furthermore, it is required to identify the suitability and maturity of the different available solutions. This implies Research Question 1.1.

Monitoring the execution times and the hardware utilization of serverless functions is not straight forward on edge devices. While, for instance, AWS Lambda provides statistics about these runtime metrics, AWS Greengrass does not, as it was shown in Chapter 4.3.4, out of the box, provide statistics for the function runs. Due to the resource intensity of most machine learning algorithms and the limited hardware availability on edge devices, there is a clear demand for monitoring serverless functions on edge devices. Research question 1.2 deals with the exploration of the means and metrics for monitoring the performance of serverless functions on edge devices. Research question 1.3 handles the process of collecting and persisting these metrics.

When the mentioned serverless edge computing frameworks are used for computationally intensive tasks like for example distributed machine learning, several practical questions arise. Due to the natural constraints of low energy footprint edge devices, computation resources are limited and scaling functionalities, as found in clouds, are not provided. For example, an edge device often might run out of resources, while clouds do typically not have this problem. Baresi et al. have shown that edge devices can reduce the request latency due to shorter network distances if the edge is not computationally overloaded [BMG17]. As soon as the edge is overloaded its faster to use Fog Computing nodes but even there, after some point overloading occurs and cloud data centers respond quickest [BMG17].

When an edge device has no free capacity, which might happen in real-world setups, it would, according to Baresi et al. , make sense to send requests to the cloud directly [BMG17]. If a device has free capacity, it would be useful to process the data locally [BMG17]. If forwarding requests from an overloaded serverless edge device is also feasible is researched in this master thesis.

AWS Greengrass supports the execution of local functions. Though there is currently no way to dynamically route the payload of a function call to the cloud when the edge device itself does not have enough free computational capacity. A function that decides when to execute a function locally and when not, could be implemented as a machine learning problem and leads to Research Question 2. If training data turns out to be available this could be an interesting approach.

It is currently still unknown, whether the method of forwarding requests from an overloaded edge device can improve the performance of the edge. Due to the limited research publications and the currently, unknown outcome of this approach, it is important to discuss the outcomes of the experiments by answering Research Question 3. Its even unclear if an implementation of such an approach is even feasible and possible with a given serverless edge computing framework. So one of the problems is also the identification of a mature serverless edge computing framework and studying if this framework provides interfaces to implement such an approach.

1.3 Research Questions

In order to discuss the given problems and to pursue the concept of serverless edge analytics, the following research questions were found to be interesting during the proposal finding phase, were presented to and approved by the Dean and should be covered and discussed in this thesis. Three of the research questions (RQ1.1, RQ1.2 and RQ1.3) handle theoretical foundations, while the two other questions deal with experimentation and testing of the performance of a serverless edge analytics approach.

RQ1.1: What are suitable frameworks for serverless edge analytics? Which parts of the suitable frameworks need to be used and which modifications are required to support machine learning on these platforms?

RQ1.2 What are means and suitable metrics for monitoring the performance and Quality of Service (QoS) of a serverless edge computing function? How does this differ from ordinary serverless functions?

RQ1.3: How can serverless edge computing devices and the corresponding function runs be monitored and stored (e.g. in AWS Greengrass)? Which tools are required?

RQ2: In which ways can a serverless edge computing framework decide where to execute a function (local or cloud) given the current utilization (CPU, RAM, bandwidth, availability of a GPU), QoS (latency, processing time, metrics from RQ1.2)?

RQ3: To what extent does edge analytics improve the latency and overall processing time of serverless (edge) functions? In which ways does the suggested approach (for selecting the destination of a function) change the performance of the function. Discussion of the results is required.

1.4 Aim of the Work

It is relevant to extend research into potential (real world) profiteers of serverless edge computing. Due to growing data amounts, edge analytics is a natural use case for this. This work should contribute to the exploration of architectures for analytic serverless edge computing functions and their properties. It is aimed to investigate on a problem that limits edge computing: computational constraints and overloading.

The aim of this thesis is to investigate on efficient ways for running serverless edge analytics. As stated in the Problem Statement above, Edge Devices can easily be overloaded and research about efficient ways of circumventing this problem is needed.

The ultimate long-term vision of a serverless edge computing setup would result in a boundless system where serverless functions are executed at the cloud, the fog or the edge, depending on current system properties, to best serve end devices. The aim of this work is to investigate further on this problem and to give a base for further research if this seems feasible or not.

The aim of this work is knowledge-finding about serverless edge computing and serverless analytics. It is aimed to show if it is feasible to execute serverless functions only on local edge devices in one case or forwarding requests from the edge to the cloud in another case or exclusively using the cloud in another case. It will be shown if forwarding requests from an overloaded edge device to the cloud is useful or not.

Furthermore, due to less knowledge about the current serverless edge computing landscape, an analysis about current serverless edge computing frameworks is needed. Then, its aimed to show how to monitor the quality of service and how to select relevant metrics for those frameworks.

The final results of this thesis contribute to the understanding of the current landscape of serverless edge analytics and give some recommendations for further research.

1.5 Expected Results

It is expected to have a state of the art discussion of the technologies that are related to serverless edge analytics. A comparison of the benefits and drawbacks of available software stacks is required. Tables and Diagrams should be used where they can support the quality of the thesis.

One hypothesis is that forwarding an event to cloud data centers in case of a computationally overloaded edge could reduce the request latency. It should be shown if this is the case and if yes how much it improves! If it is not the case it should be shown if it is more efficient to send requests from the end devices to the cloud directly - without an edge device.

Beside that, studying the current serverless edge computing landscape, its frameworks, hardware, metrics and benefits is required. One mayor task of this thesis is the definition of metrics and an implementation of monitoring for serverless functions in edge analytics for one of the frameworks that are identified as suitable candidates throughout this thesis. During the progress of this thesis AWS Greengrass was identified as most mature during the comprehensive literature and online analysis.

Furthermore, it is expected to discuss the results along common edge computing use cases. It should be stated where serverless edge analytics brings benefits and where it doesn't based on the findings of the thesis.

1.6 Methodological Approach

According to Mohannad and Ayash [Aya14] there are three main research methods in computer science: the experimental, the simulation and the theoretical method. They also mention that the experimental method is used in several different fields within computer science like "*artificial neural networks, automating theorem proving, natural languages, analyzing performances and behaviors*" [Aya14]. Such experimental methods involve testing and statistical evaluations.

This work includes a prototype that measures the performance of serverless edge computing functions with a clear and structured benchmarking procedure because according to Mohannad and Ayash *all the experiments and results should be reproducible* [Aya14]. The methodological approach for the overall thesis involves the following steps:

1. A consequential, iterative literature analysis about available frameworks and concepts should be done to answer the first section of Research Questions (1.1, 1.2 and 1.3) and Research Question 2. The analysis should be based on queries of big portals of scientific publications as well as the documentation of frameworks. It should include the following topics:
   a) serverless frameworks
   b) serverless edge computing frameworks like AWS Greengrass and Microsoft Azure IoT Edge
   c) common utilization metrics and performance for cloud computing and serverless functions
   d) an analysis of possible machine learning concepts that could be applied with the serverless edge computing frameworks
2. Based on the gained information from the literature analysis, a superior framework for serverless edge computing should be selected based on the identified metrics.
3. To answer Research Questions 3, a proof of concept prototype shall be implemented and used for the experiments regarding the hypothesis if forwarding improves the system performance on an overloaded serverless edge node. The prototype should:
   a) show ways to use analytic frameworks within serverless edge computing functions and show the limits of doing so.
   b) acquire and monitor performance and utilization metrics for serverless analytic functions running on the selected serverless edge computing framework.
   c) be used to analyze the acquired data and discuss the suitability of this data for training a machine learning model for deciding whether, in the current utilization state, a function is better run on a given edge device or forwarded to the cloud.
4. Finally, in order to answer all Research Questions, the acquired data from the experiments will be analyzed and discussed along the findings of the comprehensive online and literature analysis.

1.7 Structure of the Work

This work is structured as follows: Chapter 2 describes relevant theoretical background in the fields of Cloud Computing, Serverless Computing, Internet of Things, Edge Computing, Fog Computing, Analytics and Machine Learning.

Chapter 3 is used to summarize and abstractly discuss Related Work in different areas of research. Those areas were sectioned in Section 3.1 about Serverless Edge Computing, Serverless Analytics in Section 3.2 and Distributed Machine Learning in Section 3.3.

The analysis including the individual methodological approaches for the five research questions that were defined in Section 1.3 is contained in Chapter 4. This Chapter contains five sub-sections that each focuses on the analysis of one of the research questions.

Chapter 5 is based on all findings above and furthermore provides various code listings for the experiments and also lists the results and answers for the defined research questions.

Finally, Chapter 6 is used for discussing the results of this work along use cases that can benefit from serverless edge analytics and is followed by Chapter 7 which summarizes the work and gives some outlook for future work.

CHAPTER 2

State of the Art

2.1 Cloud computing

Cloud Computing is a popular term that had a significant impact on many sectors of the IT-industry in recent years. Due to beneficial impacts, on scalability and possible cost reductions, it found use in many popular applications. Edge Computing is a recent extension of traditional Cloud Computing and is based on similar principles. This section covers the basic principles of traditional Cloud Computing while section 2.4 covers Edge Computing.

The National Institute of Standards and Technology (NIST) defines Cloud Computing as a model "*for enabling ubiquitous, convenient, on demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*" [MG11].

2.1.1 Types of Cloud Computing (NIST)

Cloud computing is a broad term that includes various different types of IT-services and customer needs. NIST classifies cloud computing services as three different types: Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). Each having different targeted customer groups and attributes. Furthermore, further types were recently introduced to describe a broader range of services, like FaaS or Backend as a Service (BaaS). See section 2.1.2 for details.

IaaS - Infrastructure as a Service IaaS is the broadest service type that is used in cloud computing. Consumers are not able to manage underlying hardware, servers or network configurations but are able to deploy and run software on (most often virtualized) hardware. There are only a few software constraints depending on the given hardware

environment, and IaaS-customers can even run desired operating systems on the foreign devices. IaaS provides, manage and supply the underlying processing power, storage, networks, and other fundamental computing resources [MG11].

Popular examples include: Amazon EC2, Microsoft (MS) Azure, Google Compute Engine, International Business Machines Corporation (IBM) Bluemix.

The IBM IaaS cloud, for example, supports the following operating systems and visualization technologies on bare metal or virtual servers [IBM18a]: CentOS, CoreOS, CloudLinux, Debian GNU/Linux, FreeBSD, Ubuntu Linux, Microsoft Windows Server, Citrix Xenserver, Red Hat Enterprise Linux, VMware ESXi, Vyatta, Parallels.

Their servers support various hardware configurations including so-called bare metal *Graphics Processing Unit (GPU) cloud servers* with cutting-edge Nvidia GPUs that can be used for analytical tasks [IBM18b].

PaaS - Platform as a Service PaaS empowers customers to deploy applications that are created using programming languages, libraries, services and tools supported by the provider without managing or controlling the underlying infrastructure including the operating system. The customer has control over his application and application relevant configuration settings like e.g. runtimes and environment settings [MG11].

There are many examples for PaaS including, for example, a PHP web hosting service that allows customers to deploy their applications in a given (partly configurable) PHP environment. Or a Java hosting service with a configurable server environment where Java web applications can be deployed.

When naming broader examples, even container based services could be included.

Examples of PaaS worth mentioning are AWS Elastic Beanstalk, Heroku and Google App Engine.

SaaS - Software as a Service SaaS describes another higher abstraction of cloud computing. In SaaS, applications running in the cloud and serve a given purpose. The consumer has no insights, no configuration interface (beside settings for the application logic) and no underlying server access permissions. There are no other technical interfaces than the provided software. Access typically occurs through web browsers (e.g. webmail) or customized GUI or API interfaces (e.g. Apps, REST-APIs, etc) [MG11].

Some SaaS-providers do indeed use another PaaS or IaaS Provider internally. One famous example of this is Dropbox (itself being a SaaS solution) that was running on AWS (as a PaaS) for many years [1].

Noteworthy examples of SaaS are famous apps like the Google Suite, Microsoft Office 365 and Dropbox can be seen as SaaS.

[1] https://t3n.de/news/weggang-aws-dropbox-spart-75-962140/

2.1.2 Other types of Cloud Computing (Non-NIST)

Beside the afore mentioned service types by NIST, other service types that are commonly referred to in literature include BaaS, Container as a Service (CaaS), and FaaS (very relevant for this thesis).

CaaS - Container as a Service CaaS is a recent cloud computing type that allows users to run and manage software containers on cloud hardware. These systems allow the execution of various container based formats like Docker containers. In contrast to PaaS, CaaS does not focus on specific application engines but rather on the orchestration of containers on the providers hardware. The containers are packaged by the customers and contain their own runtime and application specific engines internally.

Popular examples of CaaS include Google Container Engine (GKE), IBM Softlayer, OpenStack, Exoscale, MS Azure and Amazon Web Services (AWS) (ECS).

BaaS - Backend as a Service BaaS is used to refer to backends, mostly the ones used for mobile backends, that help developers to connect their applications to backend storage and various cloud services like push notifications. BaaS reduces the development effort of mobile apps by providing common services. In contrast to SaaS, BaaS is not only a cloud application but a complete software package for development use. Examples include Google Firebase and Backendless.com.

MLaaS - Machine Learning as a Service Machine Learning as a Service (MLaaS) is a term that is used for describing cloud computing platforms that provide users with computing infrastructure that already cover common machine learning functionality like model training, model evaluation as well as data source integrations and data preprocessing. Many of the available MLaaS frameworks give prediction results that can be bridged with internal IT infrastructure through REST APIs.

M.Ribeiro et al., proposed an MLaaS architecture in 2015 where users must "*not be concerned with implementation and computing resources, focusing mainly on the data itself*" [RGC15].

Current vendors of MLaaS include IBM with IBM Watson, Google with Google Cloud Machine Learning Engine and AWS Sagemaker. AWS Sagemaker, taken for instance, uses commonly open source tools like Tensorflow and Apache MxNet (an efficient open source library for deep learning that can potentially be used on any hardware) internally but extends the functionality with the automatic provision of Cloud Computing resources.

FaaS - Function-as-a-Service is a new term that is often used to refer to Serverless Computing and can be seen as a type of PaaS.

Current industry examples include AWS Lambda, Apache OpenWhisk, Google Cloud Functions, Fission, Iron.io and will be covered in detail in Chapter 2.2

Due to the lack of a real server environment and product-dedicated runtime engines, FaaS reduces administrative overhead and improves the efficiency of cloud hardware. Functions can typically be used for tasks that are triggered at certain events. Due to low startup times FaaS provides efficient down-scalability and for some cases an improved cost efficiency.

2.1.3 NIST Deployment Models

NIST does not only separate cloud services by type but also by the manner of deployment [MG11]. Separating clouds into public and private ones especially makes sense when strict legal restrictions of users have to be met or when scalability and latency are an issue.

Private Cloud Private clouds are only operated for the purposes of a single customer or company. It is often operated by the companies themselves but can also be outsourced to providers that comply case specific regulations. Due to bad economies of scale and the risk of over provisioning, private clouds can be more expensive than public clouds. Private clouds can still provide elasticity scaling like their public counterpart if the company size is big enough and they are generally more often found in large enterprises than in smaller or medium sized companies [MLB+11]. The main purposes of private clouds are security and compliance concerns in e.g. public and banking sectors [MG11].

Community Cloud Community Clouds are exclusively used by a specific community of consumers from organizations with shared concerns and is managed, owned and operated by members of those communities or a shared third party.

Public Cloud Public clouds are, in contrast to the prior two deployment models, not restricted in access. The cloud infrastructure is provisioned for open use by the general public. Furthermore, "*it may be owned, managed, and operated by a businesses, academical, or governmental organizations, or some combination of them*" [MG11]. Popular examples of public clouds include AWS, MS Azure and Google Cloud offers.

Hybrid Mixing any of the approaches above (public, community or private clouds) results in hybrid cloud solutions. An hybrid cloud approach can be used to combine the advantages of different deployment models. Particular information that is not required to stay in private environments could be outsourced into public or community clouds while critical information will stay in the specific private clouds (eg. privacy requirements in certain countries). Referring to an example by the NIST, hybrid clouds could be used for bursting exceptional load peaks from private into public environments [MG11] (for sure only if data constraints allow to do so).

2.2 Serverless Computing - FaaS

Serverless computing is a recent cloud computing paradigm that is mainly shaped by AWS Lambda. AWS Lambda was the first commercial platform for serverless computing and is available (including preview) since the end of 2014 [AWS14].

Due to the rise of container based technologies like for instance Docker, traditional monolithic applications were separated into smaller, stateless containers - into microservices. The biggest advantages of microservice architectures, over traditional monoliths, include the ability of scaling applications to the current demand [Rob18]. Serverless computing gets high research interest because it can be seen as a finer approach to the microservice paradigm.

Serverless Computing still involves servers. The term Serverless describes an architecture where applications are composed of triggers and functions. There are no services that are running continuously, instead if a trigger occurs, a function will be started as a service. The triggers are monitored by a serverless framework (a service itself) but this trigger-monitoring is shared between functions and independent of the runtime of each function. The software authors are not responsible for the management of underlying resources, runtimes and server hardware (therefore called serverless). They only care about their code and not about the runtime or any services that run their code.

2.2.1 Operating principle

The two main parts of every Serverless computing architecture or framework are triggers and functions. The following two paragraphs will shortly introduce both of these concepts. Paragraph 2.2.2 will describe how the execution environment of functions is loaded.

Serverless Functions Serverless Functions (or short functions) are short lived applications that are executed on demand. Unlike traditional architectures, those small ephemeral, stateless micro containers of code do not run continuously. They are deployed as software code. Serverless functions are custom logic that can be defined by a developer directly. As soon as the functionality of a function is needed, the serverless computing framework will instantly spin up a fitting runtime and will deploy the code there. The container will execute the function code as often as needed (sometimes also only once) and then shutdown the whole container again. Containers can also be started up concurrently.

Popular serverless frameworks are not using exactly the same definition for a function. Some frameworks define a function to be a directory of code that contains all required dependencies in order to make the code work and a predefined handler-file that contains the entry point into the custom logic [AWS19j]. Other frameworks do also support broader definitions of a function. Among many others, Apache OpenWhisk allows arbitrary docker containers to be used as a function [Web19c]. OpenFaas allows even arbitrary shell programs, that usually communicate via STDIN and STDOUT, to be used as serverless functions [Ell18].

Serverless Triggers Serverless computing applications are not deployed as a continuously running application. More accurately, only the serverless computing framework is running (continuously) and handles the management of invocations, triggers and execution runtimes. If a certain functionality is needed it starts more runtime containers. Moreover it handles logging, billing and configuration.

The framework continuously monitors events that can happen. If one of such events happens a trigger is fired. Triggers that start a function can be selected by the developers. The way that these triggers are defined varies between the different frameworks. Common triggers include HTTP Requests, Events from Webhooks, Message Queues, Timer Events, Uploads to collaborating services (e.g. S3 upload triggers and action), reacting to database changes (CouchDB can trigger Apache OpenWhisk to execute a function), reacting to collaborating chat services (e.g. Slack) and many others.

Serverless Applications Serverless applications are packages that are composed of the code of one or more functions and a configuration for those functions. The configuration describes when, how, by whom and with which hardware capabilities, software stacks, parameters, permissions, logging and tracing environments the functions are to be triggered and executed.

Serverless Frameworks A Serverless Computing framework listens to predefined triggers and handles the execution environment of the corresponding functions. It runs continuously while the triggered functions do not run continuously. Due to the fact that serverless functions are not continuously running services, the frameworks can utilize the resources of the hardware for many other functions too.

Platforms and FaaS Serverless Platforms, like for example AWS Lambda or IBM Cloud Functions, are hosted versions of serverless frameworks. Some frameworks build upon open source solutions (IBM Cloud Functions is based on Apache Open Whisk).

Users of serverless platforms are usually only billed for the resources that they actually use. This includes only the number of total invocations and the resources that will be used for each invocation. The monitoring of the triggers is handled by the serverless platform itself and is usually not billed. The platform will always provide enough hardware resources and deploy a function when a trigger occurs.

Literature refers to Serverless computing as FaaS [FIMS17] because users only buy function runs *as a service* by paying for the resources used for each function run. They do not rent infrastructure or permanently provisioned hardware capacity like traditional cloud applications, like for instance hosting services and continuous running Docker containers, among others, do. FaaS is said to be a more finer granularity type of Cloud Computing as it is extending the existing types of Cloud Computing (IaaS, PaaS, SaaS, Caas and BaaS - more details see Section 2.1). It still provides access to Cloud Computing resources (serverless is still running on servers) but the deployments are not running continuously, *as a service*, anymore.

2.2.2 Cold and warm starts

The cost of traditional Cloud Computing services are usually calculated depending on their allocated resources. The actual utilization is usually not included in the calculation of the price a customer must pay [Eiv17]. Serverless functions do not run continuously and their execution container will start up when a trigger requires an execution. Common implementations for serverless computing, like for example AWS Lambda, will not spin down the execution container after a certain function has finished its work always [IMS17]. If there are no invocations for some time, the container will be spun down later but the framework might first try to reuse previously started containers. This has positive effects on the resource efficiency of the provider but negative effects on the execution latency for the user. The setup of a container typically takes some time that usually adds latency to function calls [IMS17]. If a container needs to be spun up initially, it is called a cold start while a reused scenario is called a warm start:

- Cold starts: A function might start from a spinned down state where the invocation time will include the setup of the execution container. The experiments in Section 5.5 have shown that for AWS Lambda this can take up to a few seconds.
- Warm starts: The execution container of a function (function container) might also be reused. This can happen if the same hardware server or pod executes the function. Then the program instance will already be started and be ready. A warm start will naturally have less latency than a cold start.

2.2.3 Benefits and Drawbacks

Serverless computing comes with various benefits and drawbacks in comparison to other Cloud Computing types mentioned in Section 2.1.1 and Section 2.1.2.

In 2017, A.Eivy summarized that "*Serverless has the potential to be a great abstraction offering economic advantages for simple workflows, but beware of throwing everything into it too hastily*" [Eiv17].

Mike Roberts is a popular blogger and consultant in serverless architectures. He lists various benefits and drawbacks in his work about Serverless computing in [Rob18] and [RC17] that are known today. The following paragraphs list these.

Benefits Serverless computing reduces the operational complexity of software deployments. This is also one of the main benefits of serverless computing. Less administrative work and less operational management dramatically reduce the labor costs of software deployments.

Serverless brings higher resource utilization of hardware as the server hardware is only reserved for a customer service during invocations and not during idle time. This results in reduced hardware demand, energy use and reduces the environmental footprint, creating "*Greener*" computing [Rob18].

Serverless can also dramatically reduce costs for software that has inconsistent traffic requirements and only reacts to occasional events. Serverless Platforms usually only bill users for the actual invocations and not for the idle time. Furthermore, serverless frameworks handle some amount of automatic scaling and reduce the time that is needed to update server code (time to market decreases). This gives developers the ability to develop and release at a faster pace. They only need to focus on the actual business problems and not at the distributed environment.

Having quicker release cycles provides developers with the opportunity to develop and test their ideas on the market more quickly. This gives early feedback and reduces the risk during prototyping.

Drawbacks Serverless computing also comes with various drawbacks. Per design, serverless functions are started in an isolated environment for each invocation. Section 2.2.2 describes the behavior of cold and warm starts. The time that is needed to provision the function runtime has a negative effect on the overall latency of the function.

Serverless frameworks are currently not standardized. The available implementations are not interoperable and software developers are therefore threatened by vendor lock-in and loss of vendor control.

Beside vendor lock-in, serverless platforms usually generate a loss of general control. Developers have to trust the platform to not only take security measures during runtime seriously but to also trust delivering the raw software code to the platform itself. When the platform operator is a market rival, this could result in loss of control and be a business risk.

Another drawback of serverless functions is the lack of local testing frameworks. For many years, it was not possible to execute serverless functions locally. Nowadays, some frameworks allow a local execution but still lack important features like tracing and code debugging.

2.2.4 Suitable Areas of Application

As mentioned in section 2.2.3 there are both benefits and drawbacks in serverless computing. There are some areas where serverless computing is known to reduce the operational costs or improve processing time. IBM lists those suitable use cases as examples for their serverless computing environment (OpenWhisk) [IBM18d] but they are also fitting for other platforms:

Microservice While, according to literature, microservices can bring many advantages, the growing complexity of such systems is demanding for developers and increases costs. According to IBM [IBM18d], serverless computing environments reduce the infrastructural and operational complexities of microservice architectures.

Most traditional microservices run continuously in containers. Scaling those to multiple regions for handling additional load is expensive as it requires multiple services running in every region. Serverless functions can be placed in any region but will only be billed if they are actually executed (or triggered) there. According to IBM this is one of the biggest advantages of serverless functions in the microservice architecture [IBM18d].

IoT Internet of Things (IoT) applications usually handle a large amount of connected devices that generate data. Those data ingests are usually triggered on a sensor measurement or sent in batch bundles. If sensors start sending higher data rates (e.g. in natural disaster situations or at football matches in a stadium) cloud infrastructure usually does have to scale accordingly. According to IBM, "*it is difficult to build a system to meet these [elastic] requirements that use traditional server architectures. As they tend to either be underpowered, and unable to handle peak load in traffic, or be overprovisioned and highly expensive*" [IBM18d]. Serverless can furthermore reduce the communication complexity with IoT-Systems by reusing triggers and data input pipelines [IBM18d].

Backends All major serverless function frameworks provide addons for the automatic generation of REST APIs for functions. This reduces the workload for developers of mobile and API backends. Serverless functions furthermore, do not require extensive knowledge of server-side runtimes and empowers mobile developers to provide server-side logic for their mobile applications [IBM18d]

Data processing Especially for tasks that only run at specific times and not continuously, serverless functions can reduce the operational costs. When a serverless function is not triggered, it also does not cost money. Tasks that are only triggered sometimes can provide cost reductions. A common example of this is the pay check generation that a company does on a monthly basis but then with a high intensity.

OpenChecks[2] is a proof of concept of using IBM's cloud functions for processing the deposit of checks (cheque) to a bank account and shows the concept of triggering server-side logic only when needed [IBM18d].

Serverless computing is also suitable for triggers that occur on other events. IBM mentions cognitive processing of newly uploaded files as a valid example [IBM18d]. Running an on-premise server all the time to wait for new files to be processed may not be as efficient as serverless functions.

Event processing of message queues Serverless functions can also be triggered if certain events occur in various messaging queues like Kafka or Message Queuing Telemetry Transport (MQTT). The event driven nature of such messaging systems fit well with an event driven runtime. Serverless functions can reduce the complexity of event driven software and provide instant scaling capabilities [IBM18d].

[2] https://github.com/IBM/ibm-cloud-functions-serverless-ocr-openchecks

2.2.5 Vendor Comparison

Since the rise of AWS Lambda (the first Serverless Cloud Computing platform) created in 2014 [AWS18c], many vendors and open source projects have implemented serverless computing.

These available solutions can, generally, be categorized into three groups:

- Open Source Frameworks: Frameworks that can be hosted on own or public cloud hardware.
- Serverless Platforms: Platforms that provide serverless computing as FaaS. The platforms bill for concrete invocations of functions. The internally used implementations of the platforms vendors are sometimes open source (IBM Cloud Functions use Apache Open Whisk internally) and often proprietary (AWS Lambda and Microsoft Azure Functions have custom structures).
- Middleware Tools: Tools like for instance serverless.com that aim to provide abstractions over concrete serverless platforms and frameworks. They allow function code to be used on more than one cloud computing vendor.

The serverless platforms have differences in pricing, power and properties of the execution environment. The following properties are used to compare the different vendors of serverless cloud platforms that are listed in Table 2.1 (created in December 2018):

- (A) Price per 100ms in US Dollars ($) for the largest available instance based on working memory in MB.
- (B) Contingents per month (number of invocations / execution time in GB/s) that are included for free.
- (C) Maximum concurrent invocations per account.
- (D) Maximum configurable execution time for each function (time out).
- (E) Supported programming languages by the platform.
- (F) Official support for GPUs by the platform.
- (G) Maximum code size for the deployment packages (e.g. AWS Lambda uses ZIP archives as deployment packages that should contain all required code dependencies -> Size of ZIP-File).
- (H) Name of the open source framework the solution is build upon. No if the framework is not open sourced.
- (I) Sources.

Table 2.2 lists mayor open source serverless frameworks and was created in January 2019. Prices, free contingents, concurrency per accounts and other platform specific settings are not included in the table because the open source frameworks can generally be self-hosted and adapted to individual requirements if needed. The Table 2.2 gives an overview over GPU support, target environments (cloud, local), licenses and supported programming languages. The following properties are used to compare the features of the different open source solutions:

- (A) Target environment. Describes the locations and devices the framework is supposed to run on. Some frameworks are made for advanced cloud deployments in Kubernetes clusters while others can also be deployed on local standalone machines or even IoT devices like the Raspberry Pi.
- (B) Actively Maintained. Open source projects often lack maintenance. While creating this table, the code repositiories (Github) of the open source projects where checked for the last commits. The field is Yes if it was recently (in 01/2019) updated and No if there were not changes for a longer time.
- (C) License. Important for using this open source framework for commercial applications or if adaptations are required.
- (D) Officially supported programming languages by the framework.
- (E) GPU support. Lists if there has been shown official or unofficial support for GPUs or machine learning accelerators.
- (F) Sources.

Name	A (12/2018)	B	C	D	E	F	G	H	I
AWS Lambda	\$0.48*10^{-4}, **3008MB**	1mio requests, 400.000 GB/s	1000	**900s**	Python, JavaScript, Java, C#, PowerShell, Go, Ruby, and provides a Runtime API for any additional programming languages	No	250MB	No	[AWS18a] [BI17]
Google Cloud Functions	**\$0.29*10^{-5}**, 2048MB	2mio requests, 400,000 GB/s, 200,000 GHz-seconds of compute time and 5GB of Internet egress traffic	1000	540s	Python, JavaScript	No	500MB	No	[Goo18b] [Goo18d]
Azure Functions	\$0,24*10^{-4}, 1536MB	1mio requests, 400.000 GB/s	N/A	600s (unlimited with "App Service plan")	Python (Preview), JavaScript, Java (Preview), PowerShell (Exper.), C#, F#, , PHP (Experimental), Batch (Exper.), Bash (Exper.)	No	No Limit [Mal18]	No	[Mic18b] [Mal18] [McK18]
IBM CloudFunctions	\$0.34*10^{-4}, 2000MB	unlimited requests, 400.000 GB/s	1000 [3]	600s	Python JavaScript, Java, Go, Swift, PHP, Ruby, Kotlin(Exp.), any Docker actions	No	48MB	Apache Open-Whisk	[IBM18c]

Table 2.1: Proprietary Serverless Solutions

[3] "*individual increase possible, contact ibm*" [IBM18c]

Name	A(Target Environment)	B(Actively Maintained since 01/2019)	C(License)	D(Supported Languages)	E GPU Support	F Sources
Apache Open Whisk	Laptop, on-premise, cloud (Docker and Kubernetes)	Yes	Apache License, Version 2.0	NodeJS, Swift, Java, Go, Scala, Python, PHP, Ruby, Docker	No. But proposed	[Con19d] [Web19c] [Apa19]
OpenFaas	Cloud based (Docker and Kubernetes)	Yes	MIT	C#, Docker, Go, Java8, NodeJS, PHP, Python 2.7, Python 3, Ruby	Yes but only prototype [Koz19]	[oO19] [Web19b] [Koz19]
Fission	Cloud based (Kubernetes)	Yes	Apache License 2.0	Python, NodeJS, Go, C#, PHP and any Linux binary	No	[Fis19]
Flogo	Small IoT (Raspberry Pi, Edinson, BeagleBone) and Cloud based (Docker and Kubernetes)	Yes	BSD3	Go	No	[Flo19a] [Flo19b]
Kubeless	Kubernetes Cluster	Yes	Apache License 2.0	Python, Node.js, Ruby, PHP, Golang, .NET, Ballerina and custom runtimes	No	[Web19a]
FnProject	on-premise, cloud (Docker)	Yes	Apache License 2.0	Python, Ruby, Java, Go, NodeJs, Docker	No	[MW19] [Con19c]
Iron	Laptop, on-premise, cloud (Docker)	No	Apache License 2.0	Any language as a docker container. Communication with unix STDIN STDOUT and STDERR and ENVAR	No	[Con19b]
Backstage Functions	Laptop, cloud (Docker)	No	MIT	Only JavaScript	No	[Con19a]

Table 2.2: Open Source Solutions

The tables (Table 2.2 and 2.1) show that in 01/2019 there are no serverless cloud platforms that support GPUs or machine learning accelerators today. Furthermore, there are only two open source serverless frameworks that either proposed or showed some prototypes. This can severely limit the possibilities for machine learning because most of the machine learning algorithms perform much better on GPU hardware due to the significantly increased amount of cores (compared to Central Processing Unit (CPU)s) for parallel computing [RMN09].

2.3 Internet of Things and Cyber-Physical Systems

According to Minerva et al. [MBR15] it is not easy to define IoT. The definition of IoT often depends on the perspective of the viewer and various publications use different working definitions. Minerva et al. finally define IoT by nine features summarizable as:

"Internet of Things envisions a self configuring, adaptive, complex network that interconnects 'things' to the Internet through the use of standard communication protocols. The interconnected things have physical or virtual representation in the digital world, sensing/actuation capability, a programmability feature and are uniquely identifiable. The representation contains information including the thing's identity, status, location or any other business, social or privately relevant information. The things offer services, with or without human intervention, through the exploitation of unique identification, data capture and communication, and actuation capability. The service is exploited through the use of intelligent interfaces and is made available anywhere, anytime, and for anything taking security into consideration" [MBR15].

Furthermore, Minerva et al. discuss the distinction between IoT and Cyber-Physical Systems (CPS) where they note in literature that both terms are often used to describe the same conceptual idea [MBR15]. After a precise definition of both terms, they summarize that a CPS is more focused on sensor-based, autonomous, communication-enabled systems and mainly concerned about the collaborative activity of sensors or actuators. The IoT is, according to them, only concerned about identification and connection of small devices, "*things*", and "*considered to grow to the level of a CPS*" [MBR15].

2.3.1 Interaction Paradigms

There are several interaction paradigms and protocols in current IoT applications. Minerva et al. list the interaction paradigms and discuss their use cases. There is still a wide use of traditional web communication via SOAP or REST. Even though there are more specialized interaction paradigms like MQTT and Constrained Application Protocol (CoAP) that suit constrained devices and networks better. The following paragraphs provide details on the single protocols.

SOAP and REST SOAP and REST are communication paradigms that are widely used in traditional cloud computing. While SOAP is independent on the transport layer,

REST requires Hyper Text Transfer Protocol (HTTP). SOAP requests are technically XML documents and therefore transport a lot of syntactic overhead [MBR15]. However, REST is not bound to XML and can facilitate other data exchange formats but uses HTTP [MBR15]. Both paradigms are typically used for large scale applications where hard resource constraints are not given and "*a HTTP request requires a minimum of nice TCP packets or even more when considered packet loss on poor connections*" [MBR15].

MQTT MQTT is a simple and lightweight messaging protocol designed by IBM for use on constrained environments where memory and processing capabilities are limited. Core features of MQTT include a publish-subscribe messaging model with one-to-many distribution, very small headers, different delivery guarantees that can be attributed to messages, built-in support for loss of client-server connections via last wills and is, therefore, ideally suitable for use with IoT devices. Minerva et al. compare MQTT to HTTP and mention that the binary header of MQTT only takes up 2 bytes while HTTP takes much more space because the HTTP status alone is text based and therefore bigger [MBR15].

CoAP CoAP is another communication protocol that is used for constrained devices with restricted capabilities. It is conceptually compatible with HTTP GET, POST, PUT and DELETE methods and URLs for identifying resources, but unlike HTTP, is a compact binary format. Minerva et al. mention that CoAP unlike HTTP, does not have many weak clients and a few powerful servers but has usually many weak servers and few powerful clients [MBR15].

Bridges Additionally to heterogeneous communication paradigms, Minerva et al. also mention the upcoming concept of Brokers, entities that are able to dispatch messages between heterogeneous producers and consumers. Collina et al. introduced a QEST broker in 2012 where they bridge MQTT and REST with a custom service [CCVC12]. This service translates REST requests to MQTT topics and the other way around. Relating to the edge computing focus of this thesis, these mentioned bridges are often combined with edge computing nodes like the technology of the AWS Greengrass Core [Ser17].

2.4 Edge and Fog Computing

The "*Cisco Global Cloud Index estimates that nearly 85 ZB (Zettabyte) of data will be generated by all people, machines, and things by 2021, up from 22 ZB generated in 2016.*"[Cis16] Furthermore, they also expect estimated traffic handled by data centers to be as low as 21 ZB by 2021 (7 ZB in 2016).

It is clear to see that there is a significant 64 ZB (300% of processed data) gap between data that is generated and not transmitted to data centers and data that will be processed in cloud computing environments. Ciscos Global Cloud Index notes that the mentioned

gap "*will be the space for edge computing*"[Cis16] and that it is clear, that such systems will also influence how machine learning will be practiced in the future.

Edge Computing describes distributed systems where computational intelligence is moved geographically closer to data sources and data consumers. In traditional cloud computing, data is sent to centralized data centers. Increasing traffic is increasing the difficulty of those traditional approaches. Networks might not provide the needed bandwidth or applications with critical requirements on latency so they might not get the answers in time.

2.4.1 Characteristics

Certain characteristics must be fulfilled by a system in order to be classified as Edge or Fog Computing. Cloud Computing is a centralized form of provisioning computing resources. While centralized Cloud Computing is widely in use, there is a growing demand of more decentralized ways of computing. According to Shi et al. the edge of the Internet was seen as a data consumer in the past [SCZ+16]. Content Delivery Network (CDN)s are used to distribute content, that is meant to be transferred from data centers to end users, over multiple servers and regions to meet the growing load requirements of Internet users, to store the data closer to the users and to reduce the network distance that is required to transfer the data. As mentioned by Cisco, the expected amount of generated data is rapidly growing [Cis16]. The capabilities of Cloud Computing are limited by various factors like network bandwidth and network latency. Bringing computational intelligence to the edge of the Internet could reduce the overall bandwidth and latency needed. Shi et al. report that the network edge evolves from a data consumer to a data producer and consumer [SCZ+16]. New use cases arise - e.g. Connected Vehicles, Smart Grid, Wireless Sensor and Actuator Networks and other IoT and CPS systems will require quick responses from the network in order to fulfill new use cases. Two example use case could for instance be an cloudless car-to-car communication network for communicating measurements and movements and to ultimately prevent crashes or another use case, the immediate reaction to sensor measurements of health equipment in the the IoT in case of for example heart problems.

The following characteristics of Edge Computing generally describe the requirements that need to be fulfilled by an Edge Computing System in order to fulfill new use cases.

Latency and Total Bandwidth The main purpose of Edge computing is the reduction of network latency and network bandwidth between nodes [SCZ+16]. As already indicated by Cisco [Cis16] this is necessary for data intensive applications.

Proximity The main characteristic of Edge Computing is that computing should happen in proximity to data sources [SCZ+16]. Shi et al. define the edge as any computing and network resources along the path between data sources and cloud data

centers [SCZ+16]. Selecting the closed edge device for computation is an essential part of edge computing.

Mobile users Devices and users are mobile. Connected vehicles, sensors and phones are valid examples of users that quickly change their position. Edge Applications need to be aware of changing edge nodes and moving users to always satisfy requirements on proximity [SCZ+16].

Heterogeneity In contrast to Cloud Computing environments where data centers use very similar hardware and software, edge computing nodes might differ in hardware and software details. Deployments with different software versions might bring problems for other software that depends on the availability of certain libraries or features. Requirements for high network bandwidth or special hardware, like Tensor Processing Units (TPU) and other machine learning accelerators (see section 4.4.4), might not be available at every node. Furthermore there might be significant differences in costs and energy capabilities between nodes. Edge Computing has to deal with a heterogeneous execution landscape [SCZ+16].

2.4.2 Fog Computing

Edge Computing and Fog Computing are terms that, in many ways, are related. Some experts online even use these terms interchangeably. In general, the concepts involve similar distributed architectures but target different regions of the network. Edge Computing facilitates devices that are located at the very outmost edge of the network, while fog computing deals with servers and devices that are located between the edge and the cloud. Fog computing is therefore an environment with generally more stable, standardized and secure hardware and network environments than edge computing. Bonomi et al. list the following Fog Computing characteristics [BMZA12] :

- Low latency and location awareness,
- Wide-spread geographical distribution,
- Mobility,
- Very large number of nodes,
- Predominant role of wireless access,
- Strong presence of streaming and real time applications,
- Heterogeneity.

Those characteristics are similar to the characteristics of Edge Computing. The research contributions of this thesis that are focused on Serverless Edge Analytics might therefore also be relevant for Fog Computing and related research fields.

Mobile Edge Computing

The aims of Mobile Edge Computing (MEC) are to reduce latency and to ensure highly efficient network operations and service delivery for an improved user experience [HPS+15]. These are similar to the aims of general Edge Computing. The European Telecommunications Standards Institute (ETSI) is currently developing technical standards and prototypes for MEC. Mobile edge computing describes a distributed computing environment that runs at the edge of mobile broadband networks. The concept involves mobile broadband base stations that will, in future, be usable for application developers. Base stations will be able to run arbitrary software from third parties in a virtual-machine-based environment. Due to the manifold use cases including IoT, connected vehicles but also caching and augmented reality, literature lists mobile edge computing as a key technology towards 5G [HPS+15].

Additionally, "*Mobile Edge Computing (MEC) allows for the use of its services with low latency, location awareness and mobility support to make up for the disadvantages of cloud computing*" [BMG17].

2.5 Analytics and Machine Learning

2.5.1 Analytics

Finding a definition of analytics is not straight forward. Due to the manifold use of the term, a clear definition might exclude some approaches. Analytics is generally used as an umbrella term for the process of applying and developing statistical models for gaining actionable insights from data. Cooper et al. define Analytics as "*the process of developing actionable insights through problem definition and the application of statistical models and analysis against existing and/or simulated future data*" [C+12]. Traditional approaches of data analysis include qualitative, quantitative and exploratory data analysis while recent approaches focus on supervised and unsupervised model training.

2.5.2 Machine learning

Machine Learning has been a large research topic since decades [WD92]. The term Machine Learning can be introduced and shortly described as software that can reason new data points due to its ability to learn the structure of previously seen data.

In 1997, Tom Mitchell presented a very broad definition of the term Machine Learning and also provided an example to it: "*A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. For example, a computer program that learns to play checkers might improve its performance as measured by its ability to win at the class of tasks involving playing checkers games, through experience obtained by playing games against itself*" [Mit97].

In the last years, the number of publications and applications involving Machine Learning have increased. The increasing interest in Machine Learning (ML) in the last decade can be explained by the emerging performance improvements of GPUs since 2009. Raina et al. (2009) argue "*that modern graphics processors far surpass the computational capabilities of multicore CPUs, and have the potential to revolutionize the applicability of deep unsupervised learning methods*" [RMN09] because they were able to train their Algorithm 70x faster by using GPUs [RMN09]. They were able to reduce the time required for training from several weeks to around a single day [RMN09].

Those performance improvements enabled new use cases for ML. Not only are companies looking into machine learning, many researchers are currently extending the advances of Machine Learning algorithms into their expert fields.

To mention a local example of researchers that apply ML to their research area: A research team of the Medical University of Vienna has recently shown that "*Neural networks are able to classify dermoscopic and close-up images of nonpigmented lesions as accurately as human experts in an experimental setting*" [TRA+19].

Beside researchers, ML is also applied in the industry; according to Bourne Vanson 88% of enterprise IT decision makers were looking into ML in 2018 while 44% were already investing and 29% were planning to invest into the new technology [Bou18].

ML itself is a broad field with many different algorithms, implementations, hardware and general approaches for Machine Learning. In literature, ML is often classified into three approaches: Supervised learning, unsupervised learning and reinforcement learning [Gro17]. Each approach can be applied to different categories of input data and has different use cases. The next sections will shortly introduce the three approaches to ML.

2.5.3 Supervised Machine Learning

A data set usually consists of many instances that have multiple features each. These features can be continuous, categorical or binary [MKWS07]. If every instance in the data set "*is given with known labels (the corresponding correct outputs) then learning is called supervised, in contrast to unsupervised learning where instances are unlabeled*" [MKWS07].

Supervised machine learning is often used for classification and regression tasks. According to Aurelien Gron, the line between classification and regression is sometimes blurry [Gro17](P. 101). On page 9 he notes that "*some regression algorithms, [like Logistic Regression], can be used for classification as well, and vice versa*" [Gro17]. On page 101 he takes the prediction of pixel densities of a grayish image as a common example where both approaches might deliver results. He represents the pixels of the images as numeric features per instance. Each feature represents one pixel and can be an integer between 0 and 255. Taking 256 classes as a target space results in a classifier that classifies (gives the likelihood for each of the 256 classes) the grayish color. A regressor would result in a numeric value between 0 and 255 that can then be mapped to a grayish color. Such

problems can be solved with both approaches, while other problems might require only one of classification or regression.

Classification Classification is a typical supervised machine learning task [Gro17]. It is trained by features of many-many example instances along with their corresponding class and must learn how to classify new instances that come without the corresponding label [Gro17] (P. 8-9).

Regression Regression tasks are also typical in supervised machine learning [Gro17]. Predicting a target value that is not a target class but a numeric value from a given set of input features is called a regression task. Training such systems requires the input features and the predictors (the numeric target values). A trained regression model must then be able to return a fitting numeric value for instances that come without a corresponding label [Gro17] (P. 8-9).

2.5.4 Un-Supervised Machine Learning

Unsupervised learning or "*learning without a teacher*" [HTF08] addresses the task of inferring a function from unlabeled data in order to find hidden structures. In contrast to supervised and reinforcement learning the task has to be done without providing correct classes or any computable degree-of-error of the data points that are used for the training process [Gro17].

In contrast to supervised machine learning there is no labeled corresponding true output given to the input features. Unsupervised machine learning algorithms are typically used for clustering and anomaly detection, where finding unknown structures in data is the primary goal. Various implementations and research fields in unsupervised machine learning exist including, for example, market/customer segmentation analysis, recommendation systems [Tur02], intrusion detection in IT Security [THLL09] and many other tasks that require clustering of data.

Clustering Guha et al. defined clustering as follows: "*Given n data points in a d-dimensional metric space, partition the data points into k clusters such that the data points within a cluster are more similar to each other than data points in different clusters*" [GRS98]. Clustering is often used to detect groups of similar instances and especially useful in knowledge discovery and data mining [Gro17, GRS98]. Clustering algorithms have no input on which instance belongs to which group, it finds connections without the help of labels [Gro17].

Anomaly detection Anomaly detection is the task of identifying rare items, events or observations "*which raise suspicions by differing significantly*" from the rest of the data set [LO19]. It is often used in the context of predictive maintenance and security log-analysis.

2.5.5 Reinforcement Learning

Reinforcement Learning is a mix between supervised and unsupervised learning. During training, the correct "*true*" classes are not available in the given training set. During reinforcement learning, the learning algorithm gets feedback about its results - if they are correct or not. There is no information on how to improve the results or what the correct answer is. The algorithm has to find the correct answers and the incremental improvements towards a good answer on its own [Mar09].

2.5.6 Out-of-core and Online Learning

Out-of-core learning deals with machine learning problems where computers are not able to fit all training data in their own memory. In contrast to that, batch learning is a system that is unable to learn incrementally. Batch learning needs to be trained on all available data always and can not be adapted afterwards. This does not only take a lot of time and resources but also limits the use cases of such systems. As soon as learning is done, the learned predictor can be used for predictions but will not learn anymore [Gro17]. The process of using a trained predictor for predictions is called inferencing. Batch learning is also called "*offline learning*" [Gro17].

So called out-of-core algorithms avoid the drawbacks of offline learners. Such systems are able to use the training data incrementally "*either individually or by small groups called mini-batches* [Gro17]. Each of the small training steps involves some data instances only and is computationally less intensive for each batch than training with the complete data in one batch with offline learners. Out-of-core algorithms chops the huge amount of input data into smaller parts, loads parts of the data, runs training steps on that data and repeats the process until it has seen all available input data [Gro17].

Online learners are closely related to out-of-core learners. Both terms are used for similar algorithms. Online learners are out-of-core learners that are not only able to split the training data into little pieces but are also able to train the model (and therefore dynamically adapt) to new data as soon as new data arrives [Gro17]. Online learners are used in situations where models need to adapt to new patterns over time, e.g. in stock price prediction.

An important parameter for online learners is, according to Gron [Gro17], called learning rate. The learning rate determines how fast the algorithm adapts to new data. Having a low learning rate results in an algorithm that slowly learns from new instances and is more stable to noise of new data points. A high learning rate results in a fast learning algorithm that quickly adapts to new trends but might therefore be prone to bad data quality [Gro17].

CHAPTER 3

Related work

This chapter will summarize on the insights of other researchers in related work. This helps differentiating between approaches in the discussion section and furthermore this is necessary as a foundation for answering the research questions.

Many researchers are currently working with Serverless computing. Due to its opportunities in reduction of deployment complexity and processing costs this concept is currently used by all mayor cloud providers as this can also be seen in Section 2.2.5 by an undertaken comparison of the cloud providers. Research of Serverless computing is currently focusing on bringing Serverless functions to Edge devices too [GND17].

Still, there are yet only very few publications that combine all three main aspects of this thesis, that are, serverless functions in edge computing for machine learning tasks. Anyway, there are many publications that discuss relevant concepts in either one or more directions of this thesis. The following sections will summarize on relevant findings. For easier reading, the papers are categorized into the following categories:

- Related Work on Serverless Edge Computing (Section 3.1)
- Related Work on Serverless Analytics (Section 3.2)
- Related Work on Distributed Machine Learning (Section 3.3)

3.1 Serverless Edge Computing

Deviceless Edge Computing: Extending Serverless Computing to the Edge of the Network Glikson, Nastic and Dustdar introduced the novel computing paradigm *Deviceless Edge Computing* [GND17]. They propose the extension of the Serverless paradigm to the edge of the network due to various benefits including a "*seamless*

hosting and execution environment, making it easy to develop, manage, scale and operate" [GND17].

Before "*Serverless techniques, such as sand-boxed execution of tenant-provided code and programmable mapping between event sources and actions, can be applied at the Edge*" [GND17], the authors state that many design assumptions of Serverless computing need to be relaxed in order to tackle the inherently different nature of Edge systems including "*high complexity, labor intensive lifecycle management and high costs*" [GND17].

In 2016, the authors conducted a series of experiments to find the key requirements of Deviceless Edge Computing (see Related Work "*Towards Deviceless Edge Computing*" below). Furthermore, they identified a number of challenges that need to be solved to build successful Deviceless Edge Computing Systems. These challenges include resource pooling and elasticity, security, provisioning and management at scale and others [GND17].

Towards Deviceless Edge Computing Nastic and Dustdar continue their previous work on Deviceless Computing [GND17] and further describe their novel vision of Deviceless Computing [ND18]. They found that available serverless edge gateways like AWS Greengrass are in early stages and still face many of the identified challenges from 2016 (see paragraph above) [ND18].

In their publication they approach to a "*full stack platform for supporting executing and automatically operating Deviceless applications across Cloud and Edge in a unified manner*" [ND18]. To do so, they present a generic reference architecture of a Deviceless Platform and envision that time-critical data should in future be processed on Edge devices and selected data should still be forwarded to the Cloud for a further, more powerful analysis and long-term storage.

Their model abstracts the concept of traditional triggers in Serverless as so called *Intents*. Each *Intent* is a data structure that describes a task that can be performed in a physical environment and contains various meta information (like Privacy, Quality, Costs, Security Restrictions and Configuration Parameters). If an *Intent* is triggered, a "*suitable task/function is dynamically selected*" [ND18]. Furthermore, each *Intent* references a developer-defined *IntentScope* that is used to delimit the range of an Intent [ND18]. An *IntentScope* helps the runtime to select the best matching task instance if there exist multiple available *IntentScope* implementations.

Beside that, they describe other parts of their reference architecture including a provisioning middleware that runs as well on Edge Devices as in the Cloud and is based on their previous work on Edge middleware infrastructure [NTD16].

Empowering Low-Latency Applications Through a Serverless Edge Computing Architecture [BMG17] Baresi et al. pick up the concept of deploying a serverless architecture on edge devices, to reduce the overhead that comes with "*always-running VMs/containers*" [BMG17], which is, due to the fact that these nodes can not "*scale infinitely*", necessary. Cloud data centers have much higher computational reserves and

give customers "*practically infinite*" resources. According to them, a serverless edge computing architecture cannot be simply implemented with existing cloud technologies like adopting virtualization and containerization technologies. The resource-finite nature of edge servers constrains the possibility of deploying traditional (big and resource intensive) applications on them [BMG17].

To demonstrate that serverless edge architectures "*can outperform a typical serverless cloud provider under certain circumstances and requirements*" [BMG17], they ran experiments in three environments. On simulated MEC servers in an edge data center, secondly in the cloud and thirdly on a local device only. They simulated the MEC environment with a small cluster of 4 virtual machines with 2x CPU, 4x Gb of RAM and 100 GB SSD each in a local server center of the university. Each Virtual Machine (VM) hosted a single service of Apache OpenWhisk.

The simulation started by uploading pictures to the built-in CouchDB database instance of OpenWhisk. This triggered a serverless function that performed feature extraction and matching with a visual recognition library [BMG17].

The cloud alternative was implemented in AWS Lambda, AWS S3 and AWS Rekognition because they considered it as the first-available most mature serverless solution at that time [BMG17].

To compare the deployments they ran 100 to 1000 requests concurrently. For the case of 100 concurrent requests, the latency of the edge-based data center solution is 80% lower than the cloud-based solution. The latency of the local-only solution is another 31% lower. For 1000 concurrent requests, the results are very different. The local edge-based data center still shows 72% lower latency but the local instance did not return results because it was computationally not able to handle the load of 1000 concurrent requests. The local node overloaded.

They foresee, that for heavier workloads with many concurrent requests, the cloud solutions could outperform the edge-based solution [BMG17]. The higher latencies introduced by sending/retrieving data from/to the Cloud are compensated by the high scalability and parallelism. The cloud serves almost all requests simultaneously [BMG17]. But they also mention that they expect edge nodes to have access to more resources than in their experiment.

To summarize, the work of Baresi et al. showed that edge-based solutions could easily get overloaded. They follow, that recovery of overloaded edge severs could not only involve cloud data centers but other edge devices with free resources and short network distances [BMG17].

A Serverless Real-Time Data Analytics Platform for Edge Computing Nastic et al. continued their work on serverless edge computing and "*proposed a unified cloud and edge data analytics platform, which extends the notion of serverless computing to the edge and facilitates joint programmatic resource and analytics management*" [NRS+17].

Similar to other researchers (see above), they see edge computing as an enabling technology for many use cases where the data volume, that can not be sent to the cloud, is too high or latency restrictions are too strict. That state that especially medical use cases like measuring human vital signs will potentially benefit from edge computing.

They follow up that due to the limited computational capabilities of edge devices, serverless edge analytics could be used without traditional cloud resources but its not always ideal. As far as edge devices can deal with the analytic complexity, "*time-sensitive data, such as life-critical vital signs, can be analyzed at the edge, close to where data are generated instead of being transported to the cloud for processing. Alternatively, selected data can be forwarded to the cloud for further, more powerful analysis and long-term storage*" [NRS+17].

Their "*proposed serverless data analytics paradigm is particularly suitable for managing different granularities of data analytics approaches bottom-up. This means that the edge focuses on local views (for example, per edge gateway), while the cloud supports global views, that is, combining and analyzing data from different edge devices, regions, or even domains*" [NRS+17].

In order to reach high level objectives of edge computing like optimizing the network latency metrics, they introduced an orchestration layer that decides how to orchestrate the underlying resources and functions of the serverless edge analytic gateway. "*it can use the scheduling and placement mechanisms to determine the most suitable node (cloud or edge) for an analytics function to reduce the network latency* [NRS+17].

Another idea of them is the differentiation of critical and non-critical functions in case of heavy workloads or overloading situations of a single location.

EdgeBench: Benchmarking Edge Computing Platforms Das, Patterson and Wittie compared two of the newly available edge computing platforms, *AWS Greengrass* and *Microsoft Azure IoT Edge* by benchmarking them based on three exemplary workloads and the corresponding performance metrics [DPW18]. They also compared the performance of the edge frameworks with the performance of the vendors cloud-only serverless implementations AWS Lambda and Microsoft Azure Functions.

"*Criteria of interest include the platform architecture, programmability, performance, and cost. To quantify these differences consistently requires benchmarks of common uses cases. Further, it requires standardized collection of metrics, such as end-to-end latency, device compute time, device resource utilization, bandwidth usage, and cost. To make an informed choice between edge and cloud platforms, the benchmarks and metrics must also allow fair comparison across different types of deployments*" [DPW18].

For their edge setup they selected the Raspberry Pi 3B as their edge node. Azure Edge and AWS Greengrass both support this device.

For benchmarking the workloads, they selected three types of applications and implemented them in Python: "*a speech/audio-to-text application; an image recognition*

application; and a scalar value generator that emulates a sensor, e.g. a temperature sensor" [DPW18].

Speech/audio-to-text and image recognition applications are currently often executed in the cloud [DPW18]. The scalar application is however "*an example of an extremely lightweight workload [that] allows [...] to measure the performance of each framework when the computation and data volume at the edge are negligible*" [DPW18].

They describe their architecture as a processing pipeline: For the workload, processing is done on the edge devices and the performance metrics are afterwards sent to the cloud backend of the edge services (AWS IoT Hub and Azure IoT Hub). The cloud-only implementations also send their metrics to another cloud backend.

During execution of their benchmark applications, they collect various metrics:

- Compute time: This is the total time required at the Raspberry Pi [DPW18].
- Time-in-flight: This is the time taken for a message to reach the cloud backend (IoT Hub) after sending it from the edge device (similar to one-way latency) [DPW18].
- End-to-end latency: This is the difference between the time when the input is ingested at the edge device and the time when the final results are available in the cloud storage backend.
- Payload size: This is the size of the message sent from the edge device without the framework overhead [DPW18].
- CPU and memory utilization: The average CPU and memory utilization on the edge device over the execution of a given benchmark. For AWS they used the *top* command of Linux while for Azure they used the *dockerstats* command. [DPW18].
- Bandwidth: They also obtained the used bandwidth for the edge based and cloud-only scenarios by using vnstat [DPW18].

"*In Greengrass, the image, audio and local statistics directories for storing metric values are mounted into the execution environment as 'LocalResources'. In Azure Edge, the same directories are directly mounted as volumes in the Docker container*" [DPW18].

The results of their benchmarking runs can, according to them, be discussed along four dimensions:

1. End to End Latency: They observed that across all benchmarks, Azure Edge has the highest end to end latency. Even the Azure cloud is significantly slower than AWS. This can be explained by the fact that the Azure IoT Hub cloud backend is queuing and batching incoming requests for several seconds [DPW18]. Another quite surprising insight is that the end to end latency is lower for (cloud-only) AWS Lambda than for AWS Greengrass for some cases. Especially when the Raspberry

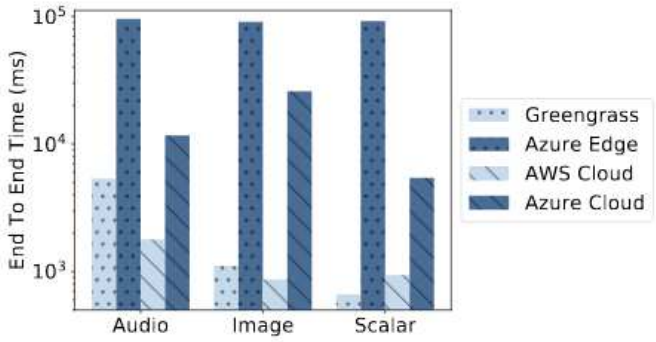


Figure 3.1: Average end-to-end latency in the edge and cloud-only pipelines for all benchmark applications [DPW18].

		Total Input Size (Mbytes)	Total raw Payload Size (Mbytes)	Total MBytes Transmitted in Network	
				AWS	Azure
Audio Trials = 104	Edge	8.83	0.02	0.25	0.26
	Cloud		8.83	9.06	9.09
Image Trials = 500	Edge	71.69	0.38	0.9	0.96
	Cloud		71.69	73.10	73.49
Scalar Trials = 200	Edge	0.05	0.05	0.33	0.26
	Cloud			0.47	0.38

Figure 3.2: Total input, payload, and actual data transmitted in edge and cloud only deployments for the three benchmark applications along with number of trials [DPW18].

Pi 3B is struggling with the load of the task (it did that for speech processing). The tasks with low CPU use like generating random scalar values for simulating sensor values had lower end to end latency on the edge pipeline with AWS Greengrass than with AWS Lambda. Figure 3.1 shows the average end to end latency of the four different pipelines.

2. Bandwidth Utilization: By using vnstat they measured the used bandwidth of the four different deployments. They observe that there is drastic reduction in the per message size in the edge deployments compared to cloud [DPW18]. Figure 3.2 shows a Table from Das et al.'s work where the bandwidth utilization of each pipeline can be seen.

3. Local Resource Utilization: To measure the local resource utilization of the edge nodes (AWS Greengrass versus Azure IoT Edge) they looked at the average total CPU and memory percentage used by all of the containers running specific to Azure Edge for Azure Edge and looked for the total average CPU and memory percentage usage by all processes under the Linux user for AWS Greengrass *ggc_user*. The

observed that "*the CPU % of Greengrass and Azure Edge are very similar, though the RAM consumed in Azure Edge is always higher*" [DPW18].

4. Infrastructure Cost: The costs of AWS Greengrass are minimal because they only involve the costs for the Greengrass Core itself plus the storage costs for the cloud backend. Compared to a cloud-only approach with AWS Lambda, the costs are around 7 times higher there than for Greengrass. They note that especially for applications with for example many video cameras, the costs for a cloud-only approach would quickly add up. For simplicity they left out the cost estimations for Microsoft Azure [DPW18].

To summarize, their "*results show that the performance of Greengrass and Azure Edge are comparable, with the exception that Azure Edge exhibits higher end-to-end latency due to its batch-based processing approach. Further, [their] results show that for the image and scalar pipelines, the performance of Greengrass is comparable to that of the AWS cloud-only pipelines, while reducing the network bandwidth usage. These results indicate that edge computing is a promising alternative to cloud computing for CPU light workloads*" [DPW18].

3.2 Serverless Analytics

Serving deep learning models in a serverless platform Ishakian et al. have recently evaluated the suitability of a serverless computing environment for the use of inferencing with large neural network models by utilizing AWS Lambda and the Apache MxNet deep learning framework [IMS17]. Their main goal is to understand if a serverless computing platform can be utilized for neural network inferencing. They mention that training neural networks is a "*time consuming job that typically requires specialized hardware, such as GPUs* [IMS17]. But, only using a trained neural network for inferencing (and not training it) requires far less computational power and less time [IMS17].

They evaluated three popular image recognition models where the neural network models defer in size. They range from small (5MB) to very large (500MB). To factor out delays that might occur when the Lambda function needs to download the model on each invocation, they bundled (included in the deployment zip archive) the model files with the AWS lambda function and its dependency libraries.

To measure the suitability of AWS Lambda, they measured their function runs under the two particular situations. Cold starts and warm starts. Those have particular influnce on the latency of serverless platforms. Their metrics are the response time (user observed latency), prediction time (model execution time) and cost (total cost of executing the AWS Lambda function).

Their results show that the total delay and prediction time of the neural network models decreases when the memory size of the Lambda function increases. The size of the memory

of AWS Lambda functions ranged from 128MB to 1536MB during their experiments in 2017 [IMS17]. While they already suspected those results, they found that, as soon as the Lambda function is configured to use more than a certain amount of memory, their code does actually not use all of the available memory. They assume that the cause thereof is the resource management of AWS Lambda: While the working memory is configured by the FaaS developer, the CPU power, disk I/O and network bandwidth are not configured by the developer, they are automatically chosen by AWS Lambda and are proportional to the chosen memory settings [IMS17, AWS17].

The interesting observation of them is that while increasing the memory does also increase the costs of the function, the total execution time does not increase enough to offset the cost [IMS17]. An increase in memory size does not necessarily lead to improvements in performance [IMS17].

The authors also mention that the measured latency of warm starts was in a valid range. But the latency of cold requests was sometimes significantly higher. They mention that this might impact Service Level Agreements (SLAs) badly. To improve performance and reduce delay, they recommend providing access to GPUs. Furthermore, they see the limited (temporary) disk space of AWS Lambda functions - 500MB - as a risk for bigger neural network models [IMS17]. Bigger models can not be used on AWS Lambda.

Serverless Computing: One Step Forward, Two Steps Back Hellerstein et al. address critical gaps in "*first-generation serverless computing* and discuss why "*serverless today is too less*" [HFG+18]. According tho them, "*Current serverless offerings are a bad fit for cloud innovation and particularly bad for data systems innovation*" [HFG+18].

Beside listing use cases that are suitable for a serverless architecture they compare various projects that have been implemented with AWS Lambda and state that applications that are dealing with independent requests - requests that do not require to communicate with other services - can directly benefit from Lambda's auto-scaling features [HFG+18]. An example for this is image recognition (inferencing).

But they list many drawbacks and bottlenecks of current serverless platforms. The limited lifetime of the functions and warm/cold start behavior requires developers to assume that the state will not be recovered across invocations. They cite a study that shows that AWS Lambda appears to be very limited in computational power and network bandwidth internally because functions of different users are usually executed together on one AWS VM [HFG+18, WLZ+18]. Beside those constraints, they list missing support for specialized hardware like GPUs and communication through slow storage (data is uploaded to S3, the first function triggers due to the upload and processes the data from S3, the function puts the results back to S3, another function triggers due to the second upload to S3, the second function needs to load the data from S3 again) as mayor drawbacks.

To evaluate the severity of the mentioned drawbacks they documented three case studies. One case study observed the behaviour of Lambda for model predictions. Due to the

lack of GPU access, the requests were significantly (27x) slower on Lambda than on a m5.large EC2 instance. They also compared the costs and found that AWS Lambda with triggers on AWS SQS costs more than having a ZeroMQ instance in EC2. They compared running 1 million messages per second and the EC2-ZeroMQ based solution is 57x cheaper.

Another case study explores AWS Lambda for model training. Each Lambda was used to run as many training steps during its maximum lifetime of 15 minutes. Running their training data set (90GB in size) took 465 minutes in total on AWS Lambda and costs $0.29. They ran the same experiment on a m4.large instance with 8GB of RAM and 2 CPUs. The AWS Lambda function was limited to 640MB of RAM. The AWS EC2 experiment resulted in a total training process of 21.6 minutes and only costs $0.04. Each training iteration on EC2 is significantly faster because the data can be loaded from the EBS volume of the machine. The Lambda experiment had to load the training data from S3 and this was much slower. They also compared the time that was needed to run one learning iteration: 0.01 seconds on EC2 and 0.59 on Lambda. Summarizing: Their "*algorithm on Lambda is 21 times slower and 7.3 times more expensive than running on EC2*" [HFG+18].

3.3 Distributed Machine Learning

Deep Osmosis: Holistic Distributed Deep Learning in Osmotic Computing Morshed et al. list various current challenges of a holistic view on Deep Learning in Osmotic Computing. They present their vision of a "*distributed deep learning approach for cloud, edge, and mobile edge environments*" [MJS+17] and discuss the problems that could arise with this concept.

In order to discuss the challenges in realizing an edge-cloud-driven distributed deep learning approach, they use a medical scenario as a reference sample [MJS+17]. They see electronic health records as a consequentially appropriate use case for local deep learning because the records are generally stored locally within the clinic and are only shared upon return requests due to privacy concerns [MJS+17].

They state that developing such an holistic distributed deep learning approach comes with several complications and challenges including:

- "*Heterogeneity in the underlying datasets introduces significant complexity in training and developing deep learning models*" [MJS+17]
- "*Different terminologies have been used to express the same context by different deep learning model*" [MJS+17]
- Deploying distributed deep learning over multiple edges brings challenges with configuration, deployment and combining of the models [MJS+17]

- Difficulty to determine which deep learning frameworks are suitable for distributed deep learning in osmotic computing [MJS+17]
- Models of a particular institution might not be accessible by another. "*Therefore, developing an integrated, centralized deep learning model may no longer be feasible. It may make sense to share the outcomes rather than the model used for computation*" [MJS+17].
- Learned semantics might differ between organizations and edge nodes. Standardizing and matching the parameters of different deep learning models in order to plug different models in and out is needed [MJS+17].
- Model interoperability might be difficult due to privacy restrictions between organizations [MJS+17].
- "*Deep learning has been successful in the domains of computer vision, speech recognition, and natural language processing*" because of the huge volume of available training data [MJS+17]. But in medical environments, patient data is limited and many institutions don't publish open data.
- Data Quality like "*highly heterogeneous, ambiguous, noisy, and incomplete*" data creates significant challenges. A critical challenge is to have good meta data that completes the data. The machine learning algorithms should consider "*data sparsity, redundancy, and missing values*" [MJS+17].
- Semantic ontology and expert knowledge can significantly increase the accuracy of the output of deep learning models. Public databases, like DBPedia, and Ontologies, like the Gene Ontology Project, could be included into the reasoning process [MJS+17].
- "*Deep learning models cannot handle temporality in data. For example, [... due to natural disasters, the data might be...] affected in this year. Existing deep learning models cannot handle the emergence of new knowledge influenced by time factors*" [MJS+17].
- Medical data is extremely private information [MJS+17].
- Understanding the performance requirements of the model in order to select fitting nodes between mobile, edge or cloud is important [MJS+17].

They also note that machine learning involves "*four steps: data harmonization, representation of learning, model fitting, and evaluation*" [MJS+17].

When deep learning meets edge computing Due to the success of deep learning for applications like facial recognition and human tracking camera networks as well as the success of edge computing in reducing the demand for bandwidth while decreasing latency, Huang et al. proposed "*Edge Learning, a complementary service to existing cloud computing platforms, seeking to combat the challenges in crowd sourced deep learning applications. Edge learning performs data pre-processing and preliminary learning at the edge of the network; the raw data in the local regions are processed on edge servers to reduce the network traffic, so as to speed up the [deep learning] computation in data centers*" [HMF+17].

As part of their work they implemented an edge learning prototype and they state that "*the edge learning design reduces the network traffic by 80% and the running time by 69% over state-of-the-art cloud-based solutions*" [HMF+17].

The architecture of the prototype consists of three major components: "*end user devices, edge learning servers, and deep learning clusters on a remote cloud. In the edge learning framework, end users devices, e.g., mobile phones [...], crowd source the data, which can be noisy and highly redundant. The edge learning servers gather the massive raw data from end users and perform pre-processing and preliminary learning techniques, so as to filter out the noises and extract key features of the raw data. The deep learning cluster, equipped with powerful and scalable GPU resources, executes the deep learning tasks*" [HMF+17].

The edge based pre-processing is done with Principal Component Analysis (PCA) which is known to efficiently reduce the dimensionality of a data set [HMF+17].

"*Principal Component Analysis, or PCA, is a dimensionality-reduction method that is often used to reduce the dimensionality of large data sets, by transforming a large set of variables into a smaller one that still contains most of the information in the large set. Reducing the number of variables of a data set naturally comes at the expense of accuracy, but the trick in dimensionality reduction is to trade a little accuracy for simplicity. Because smaller data sets are easier to explore and visualize and make analyzing data much easier and faster for machine learning algorithms without extraneous variables to process* [Jaa19].

They tested their prototype against the famous MNIST data set for handwritten digit recognition. They found that "*with larger data set size, the learning accuracy gets improved. Although the PCA pre-processing at the edge servers lowers the accuracy, the accuracy can be improved with more training data, and the running time has a significant reduction*" [HMF+17].

Scaling Distributed Machine Learning with the Parameter Server "*When solving distributed data analysis problems, the issue of reading and updating parameters shared between different worker nodes is ubiquitous*" [LAP+14]. In 2014 Li et al. proposed a parameter server framework for distributed machine learning problems [LAP+14]. They distribute both the training data and the workloads over worker nodes while server

nodes maintain globally shared parameters of the trained model. It is different to the approach of the paper by Huang et al. (above) as it is not only pre-processing data at decentralized nodes but it is actually doing the machine learning training there. This approach does not require uploading any training data to centralized cloud services.

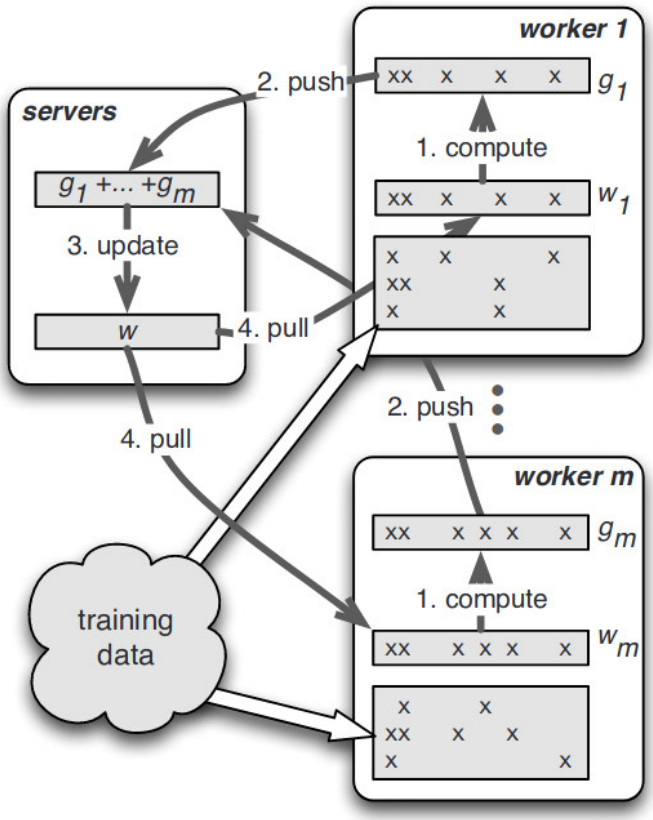


Figure 3.3: "*Steps required in performing distributed subgradient descent. Each worker only caches the working set of w rather than all parameters*" [LAP+14]

They state that the quantity of training data can range up to 1 Petabyte and results in complex models with 10^9 to 10^{12} parameters. Due to the nature of those models, their parameters are changing frequently. Changing the parameters requires an enormous amount of network bandwidth.

Problems arise because many machine learning algorithms are often sequential and synchronization adds latency to the overall processing time [LAP+14].

Furthermore, cloud environments are often unreliable and machines or connections can fail [LAP+14]. Fault tolerance is therefore a critical requirement for a distributed machine learning system.

Li et al.'s work is not focused on edge computing. Most of their experiments involved up to 1000 machines with 10GB Ethernet and 192GB DRAM each.

But the properties of their parameter server involves many techniques that are also popular in edge computing. The communication between workers and the parameter server is asynchronous, the system is fault-tolerant and provides elastic scalability.

"*A more detailed model typically improves accuracy, but only up to a point: If there is too little training data, a highly-detailed model will overfit and become merely a system that uniquely memorizes every item in the training set* " [LAP+14].

Figure 3.3 shows the architecture of Li et al.'s work and the steps that are required in performing distributed subgradient descent.

Hypothesis Transfer Learning for Efficient Data Computing in Smart Cities Environments "*Recent forecasts envisage that the number of connected things will exceed 7 trillions by 2025, resulting in an equivalent density of about 1000 devices per person all over the world*" [VPC16b].

Valerio et al. tackle the severe network congestion that is induced by the huge data amounts that will continuously be generated in smart cities [VPC16b]. They propose a machine learning approach, Hypothesis Transfer Learning (HTL), that does not require to send the data to a centralized cloud service but processes the data where it is collected [VPC16b]. They focus "*on a concrete example of activity recognition from data sampled by smart-phones in individual locations*" [VPC16b].

This technique (HTL) is different to the parameter server approach listed in the related work of Li et al. above. HTL is working on many edge computing devices in parallel on many partial models. To combine the models, all nodes communicate with the other nodes directly and without a centralized parameter server.

"*The resulting (partial) models are combined to obtain a unique model. The learning mechanisms defined by HTL make sure that each partial model is refined and improved thanks to the knowledge "already embedded" into the other partial models during their training on the partial datasets.The target of HTL is to train a final model whose accuracy is as close as possible to that of a model trained directly over the entire dataset. [...] The only information that needs to be exchanged over the network are the partial models*" [VPC16b].

For machine learning one possibility for combining partial models "*is represented by "ensemble methods" such as bagging, boosting and mixtures of experts. [... While they] might be suitable for distributed learning in parallel networks, however they are generally designed within the classical model for supervised learning and fundamentally assume that the training set is available to a single coordinating processor*' [VPC16b].

The HTL algorithm that is used by Valerio et al. is GreedyTL [VPC16b]. HTL is used to overcome the problem where some local machine learning models have enough data to train an accurate model while others don't have enough training data (see step 3 below).

The training process of HTL in a network of separate worker nodes (e.g. edge devices) involves 5 steps:

- Step 1: Each machine at each location trains a model (in their case an Support Vector Machine) on its local data.
- Step 2: All models from step 1 get synchronized. "*All the models learned at each location are sent to all the other locations. [...] Note that at every location, models are stored in the same order, i.e. model[0] is the same model at each location*" [VPC16b].
- Step 3: They run the GreedyTL algorithm on the local data at each location. But this time they are not only providing the algorithm with the local data but also with the models of the other devices that were received in the previous step. GreedyTL performs HTL and therefore overcomes the problem of inaccurate models from step 1 that can be caused by nodes that lack a huge variety and amount of training data. In order to accomplish this, GreedyTL incorporates the models of other nodes (source models) into the current local model (target model). The target model of GreedyTL can be described as a standard linear classifier with an additional term (beta-coefficients) "*that permit to include the knowledge of the source models into the target model*" [VPC16b]. When GreedyTL finishes, it results in a vector that contains the weight vectors of the local model and and beta-coefficients that "*control how much each source model can affect the behavior of the target model*" [VPC16b]. In order to find the best parameters for the weights and the beta-coefficients, the linear equation is treated as an optimization problem.
- Step 4: After step 3, there is another synchronization phase similar to step 2. Every node sends his model to all other nodes.
- Step 5: "*Once all the models have been received, each location aggregates all of them into a single model*" again [VPC16b]. While the aggregation could be done with several ways, the authors selected the majority voting and a consensus approach to aggregate the individual models.

In order to validate their approach, they tested their algorithm with a dataset that is 103MB in size. "*In order to have a clear idea about the real improvement contributed by each step of [their] procedure, [they] present separate performance values for each step*" [VPC16b] and compare their solution to a centralized solution i.e. a SVM trained on the entire dataset.

During step 1 they observed the worst accuracy, which is due to the fact that "*in the local data there are not enough examples to learn a model with good generalization abilities. After the models' exchange at Step 2 instead, [they] can see that the HTL algorithm is able to well exploit the knowledge learned at each location and learn a model with better performance. Note that each model of step 2 has been trained on the same local data*

on which was trained the models of step 1. After the second exchange of models, the performance of [their] solution are even better" [VPC16b]. After the second exchange in step 5, all models perform equally well because all of them have the knowledge from all nodes.

They compared their results to a the centralized cloud approach: The cloud approach is better in accuracy but in absolute terms the accuracy of GreedyTL is anyway very good and furthermore uses less bandwidth [VPC16b].

So finally, "*the accuracy of [their distributively trained] recognition [model] is comparable with that of a centralized solution, but the network overhead is reduced up to 77%* [VPC16b].

Accuracy vs. traffic trade-off of learning IoT data patterns at the edge with hypothesis transfer learning Valerio et al. continue (they have already applied hypothesis transfer learning to the case of distributed learning in IoT environments in the paper that is summarized above [VPC16b]) their work on hypothesis transfer learning by investigating on the traffic trade-off of centralized versus decentralized machine learning approaches in the IoT in more detail [VPC16a].

This paper is built upon so called *data collectors* that are located close to sensors and devices, similar to the data gravity concept of edge computing [VPC16a].

Their data collectors are spread across various locations and hypothesis transfer learning is used to exchange and combine partial models and obtain a unique model [VPC16a]. "*In general, in hypothesis transfer learning, instead of training a model on the whole training set, multiple parallel models are trained on disjoint subsets, and then the partial models are combined to obtain a single final model*" [VPC16a]

"*Hypothesis transfer learning methods are machine learning algorithms that have the objective of finding a profitable way of exploiting the knowledge acquired by a model m 1 trained on a dataset D_1 , in order to boost the accuracy of the learning process of a second model m_2 during its training phase on a different dataset D_2. In other words, the idea of hypothesis transfer learning is to transfer the knowledge acquired by m_1 to the model m_2 in order to improve the performance of the latter.*

In this paper they again (see previous paper) use GreedyTL as their hypothesis transfer learning algorithm.

This time they focused on the trade off between accuracy and network traffic. Their results show that on one hand the accuracy of their solution "*does not significantly vary with the number of data collectors*" [VPC16a] but that their solution does on the other hand drastically cut down the network traffic compared to a centralized solution [VPC16a].

Communication-efficient learning of deep networks from decentralized data McMahan et al. introduced the term Federated Learning by presenting an alternative to

centralized machine learning by leaving "*the training data distributed on mobile devices, and learning a shared model by aggregating locally-computed updates*" [MMRyA16]. They present a "*practical method for the federated learning [..] based on iterative model averaging*" [MMRyA16] and an extensive empirical evaluation that demonstrates that their approach "*is robust to the unbalanced and non-IID data distributions that are a defining characteristic of this setting*" [MMRyA16].

They state that ideal problems for federated learning have three main properties: "*1) Training on real-world data from mobile devices provides a distinct advantage over training on proxy data that is generally available in the data center. 2) This data is privacy sensitive or large in size (compared to the size of the model), so it is preferable not to log it to the data center purely for the purpose of model training (in service of the focused collection principle). 3) For supervised tasks, labels on the data can be inferred naturally from user interaction*" [MMRyA16].

They list several key properties that differentiates their solution from other algorithms. First off, it is able to deal with Non-IID distributed training data, which is great because certain edge devices will have skew data hence to the unique properties of particular users that are not representative of the whole populations distribution [MMRyA16]. Secondly, federated optimization is able to deal with unbalanced data. Unbalanced data happens when some edge nodes have more data to learn than others. Thirdly, the algorithm can be massively distributed meaning that McMahan expects that the number of devices participating in the Federated Learning system can be higher than the training examples that each of those devices sees. Fourthly, the algorithm is able to work with limited communication, offline or slow connections [MMRyA16].

They mention that this algorithm is powerful but still can not deal with a myriad of practical issues like for example complex correlations in the distributed dataset (e.g. speech recognition models trained on British and American English) [MMRyA16].

They compared two different approaches to Federated Learning: federated stochastic gradient decent (SGD) and federated averaging of model weights.

How Federated SGD works is basically that a fixed set"*of K clients, each with a fixed local dataset [are selected]. At the beginning of each round, a random fraction C of clients is selected, and the server sends the current global algorithm state to each of these clients (e.g., the current model parameters). [They] only select a fraction of clients for efficiency, as [their] experiments show diminishing returns for adding more clients beyond a certain point. Each selected client then performs local computation based on the global state and its local dataset, and sends an update to the server. The server then applies these updates to its global state, and the process repeats*" [MMRyA16].

"*Since these updates are specific to improving the current model, there is no reason to store them once they have been applied*" [MMRyA16].

"*Deep learning have almost exclusively relied on variants of stochastic gradient descent (SGD) for optimization*" [MMRyA16]. The SGD based algorithm selects "*a C-fraction*

of clients on each round, and computes the gradient of the loss over all the data held by these clients. Thus, C controls the global batch size, with C = 1 corresponding to full-batch (non-stochastic) gradient descent" [MMRyA16]. "*Each client locally takes one step of gradient descent on the current model using its local data, and the server then takes a weighted average of the resulting models*" [MMRyA16].

By adding additional training rounds at each client, before sending it to the global averaging step, Federated SGD results in their second algorithm Federated Averaging where not only the gradient but the complete resulting models are sent to the global averaging steps.

However, "*For general non-convex objectives, averaging models in parameter space could produce an arbitrarily bad model*" [MMRyA16].

They compared their algorithms by benchmarking them with various data sets including MNIST and CIFAR. "*Federated Averaging converges to a higher level of test-set accuracy than the baseline Federated SGD models*" [MMRyA16].

To summarize, their experiments "*show that federated learning can be made practical, as Federated Averaging trains high-quality models using relatively few rounds of communication*" [MMRyA16].

Anomaly detection on the edge Schneible and Lu see Federated Learning in Edge computing as an enormous innovation for anomaly detection processes as they, according to them, often occur in a variety of Edge computing fields including cybersecurity and the battlefield [SL17a].

For detecting the anomalies, they are using so called autoencoders, a set of two symmetric deep learning neural networks. The first neural net of an autoencoder encodes the input features of the system to a reduced form, then feeds those reduced values to the second neural network that tries to decode them to approximate the initial input features of the first neural network again. Due to lacking training data, uncommon samples will not be reproduced well - an anomaly gets detected. The deviation (or error) between the input to the first net and the output of the second net is used to determine if the current data sample is anomalous.

For demonstrating Federated Learning, they present an approach of deploying those autoencoders on each edge device. There, the autoencoders perform analytics and identify anomalous observations. "*Simultaneously, the autoencoders learn from the new observations in order to identify new trends. A centralized server aggregates the updated models and distributes them back to the edge devices when a connection is available*" [SL17a].

They describe their algorithm by the data parallelism model where the training data is split between the edge nodes of the network, such that each copy is trained on an independent section of data [SL17a]. The neural net is created multiple times in parallel, the resulting weights are sent to a central server for aggregation [SL17a]. "*In this way,*

the model at each edge device will have learned from the data gathered from all devices without having to transmit the full training to the central server. This greatly reduces bandwidth requirements and can occur when a connection to the central server is available" [SL17a].

To check their assumptions they ran four experiments.

- In Experiment 1 they split the data set into three parts and distribute it over 3 edge nodes. "*the performance of the merged model, i.e. the model produced by averaging the weights of Edge 1 through 3, results in perfect classification in our testing data. Surprisingly, this outperforms the model trained on the full set of data which only achieves an F1 score of 0.97788. This experiment shows that the merged model can perform as well as, if not better than, a model trained on all of the data at once*" [SL17a]. While the raw training data is 1.2MB in size, required bandwidth for redistributing the weights was only 432KB [SL17a].

- In Experiment 2 they tried what happens "*when the definition of normalcy changed over time. For this experiment, a univariate dataset was generated with a linearly separable boundary from normal and anomalous data*" [SL17a]. While first, the false positive rate increased, the merged model then performed well again. The local autoencoders are constantly able to adapt to changing data and updates can then be sent to the aggregation service [SL17a].

- In Experiment 3 "*we compared the performance of our federated model trained on data that is not independent and identically distributed (IID) to a single model trained on the full training data set*" [SL17a]. "*In this scenario, merging the federated models after fully training them produced poor results. Instead, [they] applied the federated learning technique from [MMRyA16] (that paper was summarized here, see section above) and randomly selected a fraction of the models to be partially trained, averaged and redistributed to the edge. The federated models were trained for 2 epochs between each iteration of aggregation and merged 100 times and the client fraction was held constant at 0.4.*" [SL17a]. Finally, "*the federated model performed approximately as well as a single model trained on the full training data set*" [SL17a].

- In Experiment 4 they repeated training a Federated Learning but this time with a bigger data set. Before, the training data was only 1.2MB and here it was 75MB. The model again performed well but not as well as a model that was trained on a centralized cloud server (with all training data and much higher bandwidth requirements) [SL17a].

Their experimental results show "*that a distributed approach to anomaly detection using autoencoders can produce similar results to a non-distributed model. The distribution allows most of the computation to occur in parallel on the edge devices. [...] As seen*

in the second experiment where the model trained on the first edge device produced poor results, the overall distributed model was robust to these errors" [SL17a].

In-Edge AI: Intelligentizing Mobile Edge Computing, Caching and Communication by Federated Learning Wang et al. propose to integrate Deep Reinforcement Learning techniques and Federated Learning with mobile edge systems, for optimizing computing, caching and communication [WHW+18]. But they are not using serverless functions for that. They state that fulfilling the requirements in the Quality of Experience of devices close to users (user equipment, UEs) is not trivial, even with mobile edge computing [WHW+18]. They "*use Deep Reinforcement Learning to jointly manage the communication and computation resources*" by discussing the methodology of using Distributed Reinforcement Learning. They proposed a framework "In-Edge AI' that uses Federated Learning "*for better deployment of intelligent resource management in a mobile edge computing system*" [WHW+18] and a proof of concept.

They discuss two representative use cases for deep reinforcement learning, Caching and Computation Offloading. For local edge based caching, they train a deep reinforcement learning model for deciding if a queried file should be put to the cache of the local device or not. For the computation offloading use case they try to improve the task execution experience of devices that connect to the edge node by allowing them to offload their computational load to the edge nodes via wireless connections [WHW+18].

Due to the low computational resources of local nodes, bandwidth-constraints and privacy concerns, they choose Federated Learning for Deep Reinforcement Learning. They state that Federated Learning could deal with several key challenges like *Non-Independent and Identically Distributed* data (not every node observes every possible training category), Limited communication (only the model updates are sent and only a part of all learning clients are used for updating the weights in every round - offline nodes are therefore not a show stopper), unbalanced data (some clients might have less training data than others), privacy and security (training data stays local) [WHW+18].

They built a prototype of their federated "In-Edge AI" to test their assumptions and ran it for one month including 9514 active users and compared the results with a centralized deep reinforcement learning agent [WHW+18]. Their results indicate that it "*can be easily seen that the specific performance of Federated Learning is near close to the results of centralized [approach] in terms of achieved hit rates of 3 clients (edge nodes)*" and that it performs better than completely (always) local or always offloading approaches [WHW+18]. Their results also indicate that the transmitted data is much lower with Federated Learning (FL).

"*With no doubt, FL must trade something for its advantages. Specifically, due to the fact that the coordinating server in FL only executes tasks of merging updates instead of taking over the whole training, the computation load of clients is inevitably heavier on account of the local training process. This will cause more energy cost on*" the side of the edge nodes [WHW+18].

Artificial Intelligence Processing Moving from Cloud to Edge Tractica is a market intelligence firm that "*combines qualitative and quantitative research methodologies to provide a comprehensive view of the emerging market opportunities*" [AK17] including Artificial Intelligence.

In a recent blog post, they summarize the advances of artificial intelligence on edge devices. The list various undertakings of companies like Google, Apple and Microsoft regarding this field. Furthermore, they state Federated Learning and Apples lates CoreML toolkit as samples for this area. While Federated Learning is clearly targeting the complete model training process, "*Apple's latest Core ML toolkit allows pre-trained models to run on the iPhone using local data, without needing to send data to the cloud*" [AK17].

As a market intelligence firm, they also give their estimations on the further developments. "*One of the biggest challenges and gaps in the market remains ultra-low-power applications like the Internet of Things (IoT) and sensors*" [AK17]. They name some hardware like Intel's Movidius that is targeting this space and expect more competition in the area of power efficient hardware in the future. They also state that we might see a "*direct correlation between the rise of AI [on edge devices] and [decreasing] investment in cloud-based AI hardware*" [AK17]

Due to their orientation as a market intelligence firm, its interesting that they list the availability of suitable hardware and not the availability of a well integrated edge analytics environment as the main enabler for edge based machine learning.

CHAPTER 4

Methodology and Analysis

In this chapter, a comprehensive analysis to answer the research questions (stated in Section: 1.3) is carried out based on information collected in the Related Work and State of the Art chapters. To answer the first three research questions with theoretical background, a comprehensive literature research of additional resources and some online analysis has been done. And the information of Related Work and State of the Art were used. The following Sections each shortly introduces the Methodology used and then present the findings for each research question.

Section 4.1 addresses Serverless Frameworks that are currently used for serverless edge deployments. This section provides details to research question 1.1: "*What are suitable frameworks for serverless edge analytics? Which parts of the suitable frameworks need to be used and which modifications are required to support machine learning on these platforms?*" [Section: 1.3].

Section 4.2 outlines metrics that are used in cloud, serverless and edge computing. Section 4.2.3 focuses on differences and similarities of metrics in cloud and edge environments. Together, both Sections analyzed research question 1.2: "*What are means and suitable metrics for monitoring the performance and Quality of Service (QoS) of a serverless edge computing function? How does this differ from ordinary serverless functions?*" [Section: 1.3].

Section 4.3 seeks to address research question 1.3 "*How can serverless edge computing devices and the corresponding function runs be monitored and stored (e.g. in AWS Greengrass)? Which tools are required?*" [Section: 1.3]. The section gives an overview of monitoring tools for the cloud based AWS Lambda and then analyses the built-in monitoring features of the edge based AWS Greengrass. Furthermore, the possibilities of how to extend the built-in monitoring is studied.

Section 4.4 discusses an a suitable machine learning approach to use within serverless functions. It was initially carried out for eventually predicting the metrics of a serverless

edge computing function in the context of a serverless edge computing framework but was not further pursued because there was no training data available within AWS Greengrass - details about that are given in Chapter 5.

4.1 RQ1.1: Suitable Serverless Edge Computing Frameworks

This chapter analyzes the first research question: "*What are suitable frameworks for serverless edge analytics? Which parts of the suitable frameworks need to be used and which modifications are required to support machine learning on these platforms?*" [Section: 1.3].

4.1.1 Methodology

In order to answer the question a methodology needs to be selected: Using a literature and online analysis, the suitable frameworks for serverless edge analytics, current serverless edge computing frameworks were discovered. This can be found in Section 4.1.3.

After discovery, the suitability of each framework for performing analytic tasks and machine learning on edge devices was analyzed, and therefore, means of comparison were defined.

In order to compare the suitability of the found frameworks for analytical tasks, Section 4.1.2 lists requirements of machine learning, edge computing and serverless computing. Those requirements can be used to estimate the suitability of a serverless edge computing framework for machine learning. The requirements were determined by analyzing literature and online resources about machine learning, edge computing and serverless computing.

To summarize the findings and to select the most promising serverless edge computing framework for edge analytics, a comparison table was created in Section 4.1.3. This section gives details about which frameworks are suitable for the task and which modifications could be done.

4.1.2 Requirement Analysis

Running analytical tasks on serverless edge devices comes with various challenges and there is only limited information available regarding this topic. While machine learning is usually practiced in high performance cloud data centers, edge computing is lacking the computational power and stability for most machine learning frameworks. This section lists and briefly discusses various challenges and also incorporates gained knowledge from the Chapters State of the Art and Related Work.

Edge Requirements

Details about the characteristics of Edge Computing can be found in the State of the Art Chapter 2.4.

The key characteristics of Edge Computing are:

- The reduction of **latency** and the total used bandwidth [SCZ+16].
- According to the data gravity concept, computation should take place in proximity to users and the computing entity should be aware of its own location. This is necessary to reduce the algorithmic workload in many cases because different real world locations bring different requirements. [BMZA12]
- The users of edge computing devices are mobile. They might change their location and then connect to another edge node in close proximity [BMG17]. Edge devices that see new users need to be able to cope with them, regardless if the user is there for the first time or comes back. Edge nodes will therefore need to communicate or even hand-over user information due to user mobility.
- In contrast to Cloud Computing, hardware and software are **heterogeneous** and not reliable [NRS+17]. DevOps on edge computing environments is usually error-prone and work-intensive. Furthermore, edge computing hardware is usually not as powerful as cloud solutions and capabilities at each edge computing node are limited.
- Very large number of edge computing nodes bring various challenges in state and configuration management [NRS+17]

In order to make a serverless computing framework suitable for the edge environments it needs to deal with the challenges of edge computing. Latency-reduction and hardware-heterogeneity seem to be especially important for productive deployments. Due to the rise of 5G and its advances in MEC, mobile users are also more and more important for edge computing.

Edge Requirement: A serverless edge analytic framework needs to deal with the challenges of edge computing by reducing the end-to-end latency and bandwidth of the application, reducing operational complexity, expecting connectivity issues, avoid issues due to heterogeneous hardware and expect users to be mobile.

IoT Requirements

Edge computing is said to be one of the enabling technologies for some areas of the IoT [AGM+15]. Many industries like healthcare, autonomous vehicles and industry 4.0 will obviously benefit from the main advantages of edge computing for their devices. These advantages include improved latency, offline availability and reduced privacy-concerns [SD16].

The Internet of Things and Edge Computing are two systems that will have many synergies in future. In order to find a suitable serverless edge computing framework, the special requirements that make Edge Computing more suitable for IoT workflows should therefore be taken into consideration. This thesis lists the main interaction paradigms of the IoT in Section 2.3.1. A research team around Minerva et al. mentioned the concept of *brokers* and bridges that dispatch messages between heterogeneous producers and consumers in the IoT [MBR15]. Serverless edge computing frameworks like Microsoft Azure IoT Edge, Flogo and AWS Greengrass do already implement parts of this behavior [Ser17].

While SOAP and HTTP are said to be considerable power consuming protocols, MQTT and CoAP are more likely to be found in constrained devices [MBR15]. Supporting heterogeneous client devices of various brands and producers requires the support of various communication paradigms. The communication paradigms of the IoT were already listed in Section 2.3.1.

IoT Requirement: A serverless edge computing framework is more suitable for the IoT if it supports more interaction paradigms of the IoT, especially MQTT, CoAP or HTTP.

Data Science Requirements

In order to be suitable for analytical processes, a serverless edge analytic framework needs to support common software stacks (a software stack is a collection of technologies that is used together in a project to solve a problem: e.g. PHP together with MySQL or HDFS together with YARN, Spark and MapReduce or TensorFlow together with Python and MatPlotLib) for machine learning.

In order to call a serverless edge analytic framework *suitable*, the framework should support runtimes and libraries of the common tools, software stacks and programming languages of the data science community.

Kaggle is one of the most widely used web resources for learning machine learning. It is a community of data scientists and is owned by Google. In 2017, Kaggle gave access to a study with over 16.000 respondents, that gives valuable insights into the machine learning community and their tool-sets [Kag17]: for example, "*on average, data scientists are around 30 years old, have but this value varies between countries. For instance, the average respondent from India was about 9 years younger than the average respondent from Australia*" [Kag17].

"*Logistic regression is the most commonly reported data science method used at work for all industries except Military and Security where Neural Networks are used slightly more frequently*" [Kag17]. Figure 4.1 shows the most commonly data science methods in the selected industry category "Academic" in 2017. Behind Logistic Regression, Neural Networks and Decision Trees are the most commonly used ones.

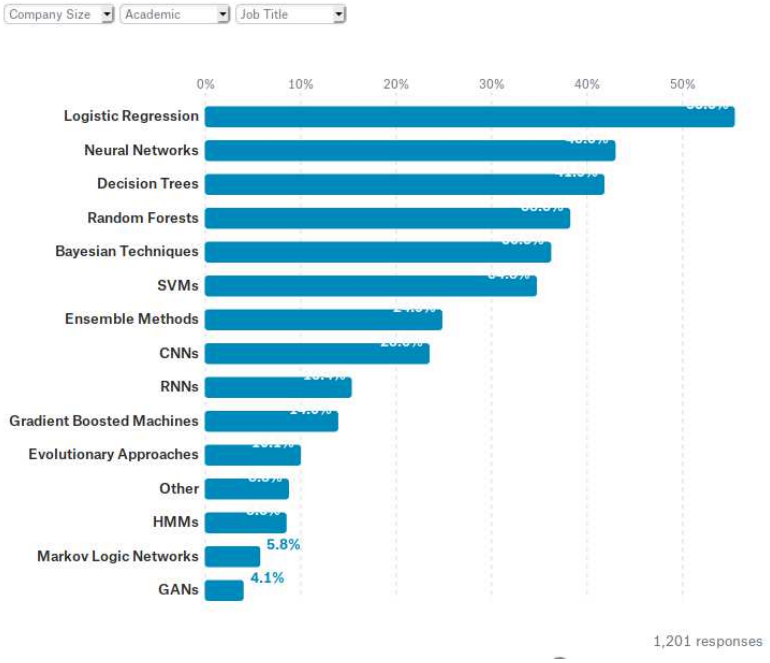


Figure 4.1: Data Science Methods used in Academic work in 2017 [Kag17]

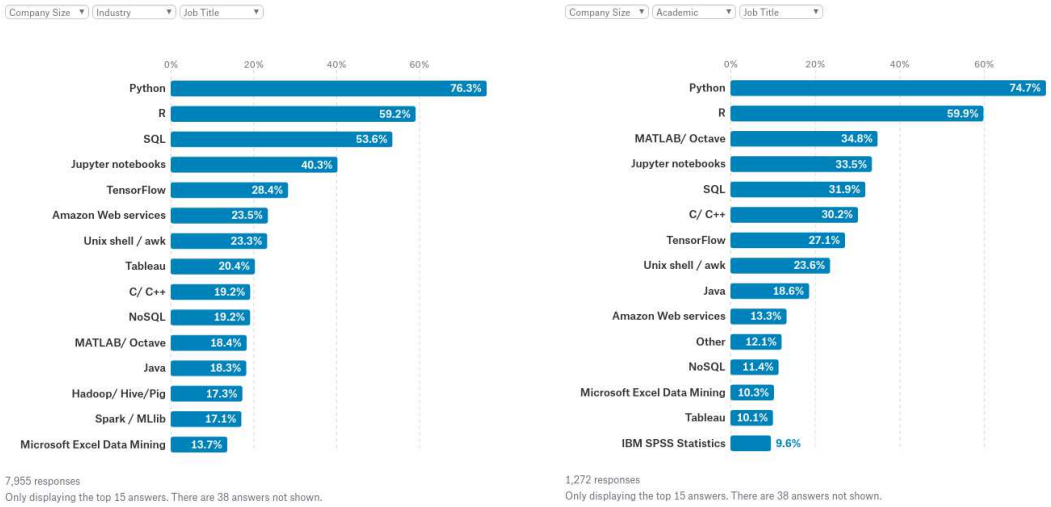
According to the study's authors, the most commonly used tool for Machine Learning is Python. Beside Python, some statisticians report using R as their most used programming language. Figure 4.2a and Figure 4.2b show the most commonly used programming languages in the categories Industry and Academic. It is clear to see that Python and R are most used in both categories and that these languages can therefore be seen as best practice and standard. It is also worth mentioning that AWS is used for many tasks in the category Industry.

The results of the study clearly show that Python is the most commonly used tool in data science. Others popular tools are R, SQL, Tensorflow by Google, Jupyter and the AWS ecosystem.

Python is also referred to as the most common language for data science and machine learning software, according to some literature [Gro17].

To verify the results of the study and to analyze the needs of the data science community, current statistics of Github were analyzed. Github hosts many machine learning projects and introduced a "*machine-learning*" tag for such repositories. Figure 4.3a of a Github-Search (on 07.01.2019) lists the most common languages of all repositories tagged with "*machine learning*" in the following order: Python, Matlab, R, Java and C++.

Requesting the same search interface of Github for the common tag "*serverless-applications*" results in the most used languages of various open source serverless frameworks. Those can be seen in figure 4.3b and include in the following order: JavaScript, Python, Type-



(a) Data Science Tools used in category Industry work in 2017 [Kag17]

(b) Data Science Tools used in category Academic work in 2017 [Kag17]

Figure 4.2: Data Science Tools used in 2017 [Kag17]

Script, CSharp, Java and Go. So, Python among with JavaScript are currently the most used languages in serverless development and Python is also the most used language in data science.

The analysis on the current data science community has shown that Python is, by far, the most used tool for data science projects and that Logistic Regression is the most commonly used approach to do it.

Community Requirement: A serverless edge computing framework should therefore support a Python runtime for data science. To make function deployments smaller, the framework should give developers access to commonly used machine learning libraries in Python (e.g. Numpy, Pandas). Including them in larger serverless function deployment packages or using them as a part of the installed Python edge-runtime (a data science python runtime) both seem to be valid approaches.

Inference Requirements versus Training Requirements

The term *Edge Machine Learning* is often used as an umbrella term for both machine learning inference and machine learning training. Inferencing is the process of using an already trained machine learning model in order to process new data points without learning from them. Training is the process of creating the model based on older data. As mentioned in Section 3.2, Ishakian et al. evaluated the stability of machine learning inferencing in a serverless environment [IMS17]. They only investigated on inferencing because training of deep neural networks is typically a time consuming task that requires specialized hardware such as GPUs which are not available in serverless environments

Languages

Python	41,413
Jupyter Notebook	34,693
HTML	16,415
MATLAB	12,997
R	4,888
Java	4,206
JavaScript	2,860
C++	2,433
C#	1,127
TeX	1,069

(a) Github search results for tag *"machine-learning"* [Git19] 07-01-2019

Languages

JavaScript	448
Python	112
HTML	71
TypeScript	56
C#	35
Java	35
Go	30
Shell	20
CSS	13
Scala	10

(b) Github search results for tag *"serverless-applications"* [Git19] 07-01-2019

Figure 4.3: Github search results for the tag machine learning and for serverless applications [Git19] 07 January 2019

yet. They mention that using trained neural networks for inferencing requires less computational power and less time than training those models [IMS17], meaning that inferencing can easier be used on resource-constrained edge devices.

The performance restrictions of many serverless computing frameworks makes machine learning training a computationally hard task there. Beside the performance restrictions, serverless functions are usually stateless, which is another challenge that serverless machine learning training has to overcome. Synchronizing distributed machine learning usually involves a lot of network traffic and the training itself requires a lot of computational power. Literature mostly refers to distributed machine learning as a task that is done in a data center where the processing nodes have high computational performance and are connected via an ultra fast network.

Bonomi et al. mention that edge and fog computing lack exactly those properties. "*It should be emphasized here that, typically there is a significant difference in scale between the fog and the cloud such that the cloud has massive computational, storage and communications capabilities compared to the fog*" [BMZA12].

Some serverless edge computing frameworks already support machine learning inference

of machine learning models that were trained on non-edge devices. AWS Greengrass ML allows the deployment of cloud trained models directly to the Greengrass edge nodes similar to MLaaS. Training them usually requires a cloud environment like AWS Sagemaker.

However, sending all training data to the cloud does not only require a lot of bandwidth and latency but according to McMahan also impacts user privacy [MMRyA16]. Storing the training data on the local edge only, will reduce the required bandwidth but increases the requirement on local storage and local processing power.

Inferencing Requirement: To use machine learning models on edge devices for inferencing, the ability to deploy those cloud-trained models to edge devices easily and without complex wrapper functions must be possible. A machine learning toolkit comparable to Tensorflow or Apache MXNet should be supported.

Training Requirement: Using serverless edge frameworks for machine learning training requires the serverless edge framework to support common training algorithms. To support such algorithms, specialized hardware, a flexible execution environment and the ability to store local training data are required. Related research has shown that Federated Learning is a promising approach for distributed machine learning in an unstable environment as edge computing [MMRyA16, WHW+18, SL17a]. Supporting Federated Learning out of the box would benefit developers.

Execution Flexibility

Analytic frameworks and machine learning require complex libraries that can easily contain multiple megabytes of code. Such libraries are usually packaged with serverless functions during deployment steps, manually or during Continuous Integration. Popular tools for managing data science libraries are package managers like Maven or PIP [Fou19, Gro17].

To package productive deployments of serverless functions with all the required libraries is a best practice because fetching the required libraries from an external server during every cold-start function invocation would take several seconds to minutes. Serverless functions are stateless! (See Section 2.2.2 about Cold and Warm starts). Warm starts are reusing old function containers, not every function invocation starts from a reused execution container of the function. During cold starts, loading of the machine learning models and libraries would occur often and waste bandwidth and increase latency. Furthermore, container-lifecycles are not under the control of the developer. They therefore need to expect, on worst case, a fresh, cold-started container on every invocation. Fetching required libraries to functions requires bandwidth and increases the latency and both is valuable in edge computing.

Packaging serverless functions with all their requirement is a challenge. Most FaaS-providers limit the code size of the functions. For instance, AWS Lambda only allows ZIP archives with a total size of 50 MB (zipped) or 250 MB (unzipped) [AWS19k]. This

limitation restricted the progress during the implementation of this thesis. Many machine learning libraries are bigger than 250MB in code size. Scikit-Learn is a popular python library for machine learning. Most applications that use Scikit-learn depend on other python libraries like SciPy that is itself based on Numpy. SciPy is a library that extends the functionality of Python with mathematical algorithms and convenience functions for working with scientific payload. Packaging all these dependencies with ones function results in a deployment archive that can easily exceed the maximum of 50MB.

Marco Lüthy blogs about a workaround in AWS Lambda where he can upload bigger archives than 50MB to Lambda by putting them into S3 instead of directly uploading them [Lü17]. While this might help for some use cases it still limits the use of more advanced analytic libraries and even with the work-around, code size is limited to 250MB uncompressed. A proper serverless framework should support functions that depend on various libraries. The current best practice, to package libraries together with code, results in huge deployment packages. Due to the fact that each deployment package needs to be loaded when the function is triggered, huge packages result in slower start up times and increase latency. To bypass this constraint, more advanced default python runtimes are thinkable solutions that are nowadays not yet implemented in any serverless edge computing framework. Current FaaS providers only support vanilla languages like Python3.6, Golang or Java1.8. Some providers allow custom runtimes, however developers need to maintain their own runtime. Providers could start adding support and maintenance for custom runtimes, e.g. a Scikit-learn runtime that has common dependencies installed and suits many machine learning developers, to reduce the complexity and code size of each function. Those runtimes could then be shared by multiple functions and would reduce the complexity for the developers.

Machine learning does not only require flexibility regarding libraries, package size and runtimes. Machine learning training is computationally expensive and often, even on strong machines or clusters, takes several minutes to hours or days. Serverless functions are often limited in their execution time (e.g. AWS Lambda is limited to 15 minutes). Having functions that are able to execute for as long as needed would be beneficial for machine learning tasks.

Execution Requirement: A serverless edge computing platform that is suitable for edge analytics needs to be flexible regarding code size, maximum execution time, long running functions and beneficially support custom runtimes or prebuilt libraries.

Active Maintainers

Serverless edge computing is still in its early days. Implementations and requirements are often changing. It is challenging for the maintainers of serverless frameworks to keep supporting new hardware, recent programming languages or other recent research findings. Many projects lack the manpower to keep up to date. Some serverless edge computing projects are already unmaintained and are therefore not a good base for further projects.

Maintainer Requirement: A serverless edge computing base should be actively maintained.

Hardware Support

As mentioned in Section 4.1.2, recent machine learning architectures, like neural nets, require special hardware like GPU for proper performance [BMZA12]. The market overview analysis of current cloud based FaaS providers that was done in Section 2.2.5 shows that no FaaS provider allows serverless functions to access GPU resources yet.

Edge devices are usually resource constrained. The default edge devices for AWS Greengrass is currently the Raspberry Pi 3 [AWS17] - a very low powered device without any hardware accelerators for machine learning. Some hardware manufacturers are working on edge devices similar to the Raspberry Pi but with additional hardware resources. Section 4.4.4 consists of a list of devices compiled through literature analysis, that are currently usable as edge computing hardware with integrated machine learning accelerators.

Hardware Requirement: A proper serverless edge computing framework should run on energy efficient edge hardware listed in Section 4.4.4 and (also) allow access to machine learning accelerators like GPUs.

Offline Model Synchronization

According to literature, training machine learning models in a distributed environment comes with a big challenge: The parameters of the federated training rounds of each device in the distributed system must be combined with the parameters from other devices in the system [LAP+14, MMRyA16].

Other researchers have solved this problem by introducing parameter servers [LAP+14]. Parameter servers are used to collect the parameters from all clients, combine them and update them on the clients again.

Due to the limited bandwidth and unstable internet connection, connecting to parameter servers can be difficult in edge computing. Serverless functions are usually stateless and will therefore require to contact the global parameter server in every round. When the edge device is offline in the moment when the machine learning functions finishes its training run, the results might be lost. To prevent this from happening, a serverless edge computing framework must be able to handle the state of the current machine learning parameters and synchronize as soon as the edge is connected to the cloud again.

Synchronization Requirement: A serverless edge analytic framework needs have to an integrated solution for state management and synchronization of the states.

4.1.3 Analysis of Frameworks

Theoretical foundations and a market overview of available serverless frameworks for cloud based serverless computing can be found in Section 2.2.

This section focuses on serverless edge computing frameworks. A number of the serverless cloud frameworks are officially available on edge devices while others can be adapted to also work outside of the cloud.

Google Cloud IoT Edge

In July 2018, Google announced the Cloud IoT Edge, a software stack that extends Google Cloud's powerful AI capability to gateways and connected devices. They also announced new accelerator hardware for machine learning called Edge TPU [Goo18a] (see Section 4.4.4).

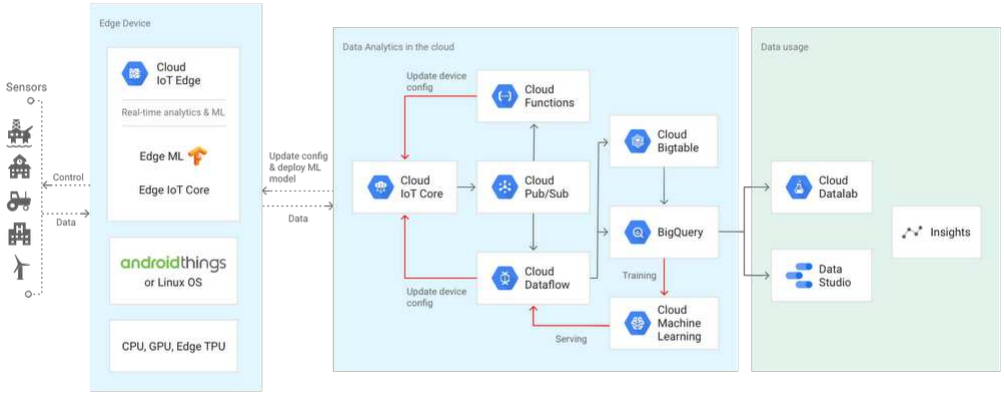


Figure 4.4: An illustration how Googles IoT Edge extends their cloud computing and machine learning stack towards edge devices [Goo18a]

Google's Cloud IoT Edge is currently supporting a local machine learning stack with Tensorflow on edge devices. Impressive demonstrations were shown at Consumer Electronics Show (CES) 2019 [1]. Google also offers a FaaS platform with Google Cloud Functions (see Section 2.2.5 for details), however Google Cloud IoT Edge is unfortunately not supporting serverless functions! Therefore, Googles Cloud IoT Edge framework can be seen as an edge analytics framework but is not suitable as a serverless edge analytic tool.

[1] https://www.youtube.com/watch?v=Ou-gulnNkaE

Project Flogo

Flogo is a lightweight, Go-based, open source ecosystem for building event-driven apps. It uses triggers and actions to process incoming events. It only supports Go for implementing actions, which are very similar to serverless functions, and its framework is written in Go itself [Flo19a]. Flogo was implemented in Go because it is 20 to 50 times lighter than Java or Node.js [Flo19a].

Due to the lightweight architecture of Flogo, it supports flexible deployments to different devices. It can run in a data center based on Docker and Kubernetes, on x86-based machines and also on Raspberry Pi and Raspberry Pi Zero devices [Flo19a]. This heterogeneity and support for low level devices makes Project Flogo a good candidate for edge computing environments and fulfills the requirement of Edge computing in Section 4.1.2.

Beside the deployment flexibility, the framework offers native, local machine learning inference with a prebuilt activity for "*TensorFlow-SavedModel-Inferencing*" [Flo19a]. It therefore fulfills the Inferencing Requirement from Section 4.1.2 but due to missing Python support, many tools of Data Scientists would not run and the Training Requirement is not fulfilled.

While inferencing is supported, there is no official documentation about supported edge devices other than Intel Edison, Beaglebone and Raspberry Pi. There is no information about GPU support or machine learning accelerators. Therefore, the Hardware Requirement be said to be partly fulfilled.

Flogo only supports Go and according to the Data Science Community Requirement in Section 4.1.2 Go was not on the software stack of data scientists in 2017 [Kag17]. The Community Requirement is not fulfilled.

Flogo is actively maintained and receives commits every other week[2], fulfilling the community requirement.

Flogo supports various triggers out-of-the-box. Default triggers include REST, MQTT, CoAP, Kafka Topics, Cron Jobs and the Flogo-cli tool [Flo19a]. These triggers support communication protocols that are very common in IoT deployments and were required by the IoT Requirement.

Furthermore, Flogo provides a graphical tool, the Flogo Flow, which allows developers to wire various triggers and actions together graphically and without code [Flo19a]. The tool converts the graphical configuration into a JSON-based configuration file for the Flogo deployment. Figure 4.5 shows a Flogo flow example configuration for an IoT Application. Flogo flow is similar to IBM's NodeRED, which is used for research on Edge based OpenWhisk functions - details on this are given in the OpenWhisk Light Section bellow.

[2] https://github.com/TIBCOSoftware/flogo/commits/master

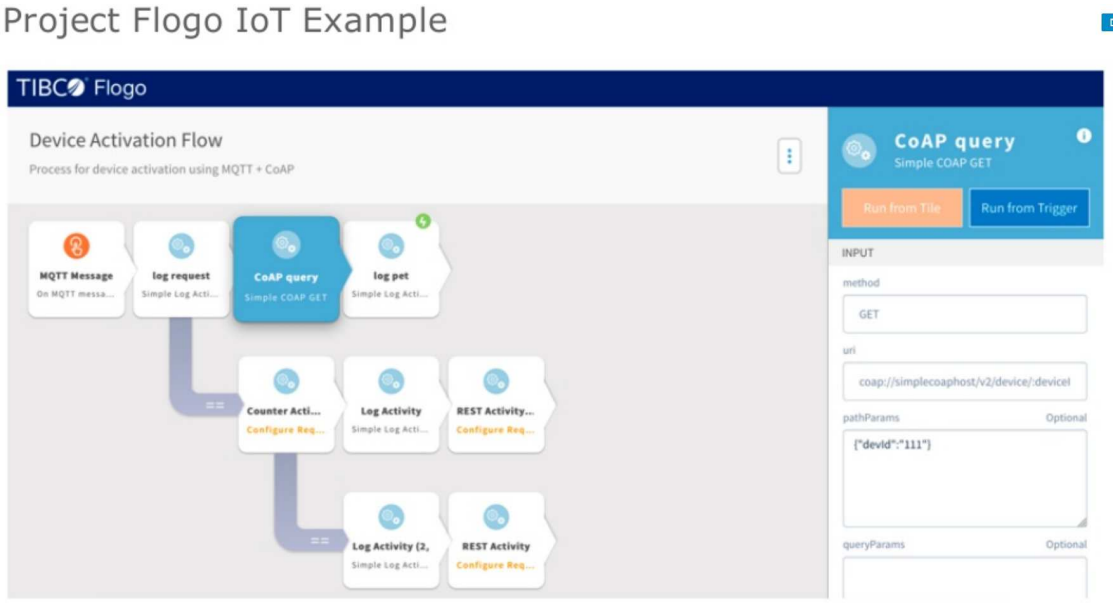


Figure 4.5: A sample flow of an IoT application in Flogo. Shown during a presentation of Kai Wähner from Tibco in 2016 [Wä16]

Overall, Flogo is a valuable platform for serverless edge computing. It fulfills many requirements for a serverless edge analytics platform and could be used as a serverless edge analytics framework for inferencing tasks.

Still, there are some requirements that are not fulfilled, as support for synchronizing the state of serverless functions is missing. Machine Learning training is not supported by the built in libraries and would be difficult to carry out due to the missing state synchronization capabilities. Furthermore, Flogo is implemented in Go and only supports actions written in Go. While Go has superior performance on small devices, and that is a clear plus for edge computing, it is currently not well supported by common data science tools.

Apache OpenWhisk

Apache OpenWhisk is a popular cloud-based open source framework for serverless computing. IBM uses OpenWhisk to run their public FaaS cloud platform *IBM Functions* [IBM19].

Figure 4.6 shows the architecture of OpenWhisk. It is based on Apache Nginx as a load balancer, CouchDB as a database and Apache Kafka as a message queue and Docker for the runtimes [Web19c]. OpenWhisk is intended to run in cloud data centers [Web19c].

Due to the mentioned architecture of OpenWhisk, which is composed from multiple large scale Apache projects, it requires a lot of computational resources. Breitgand and

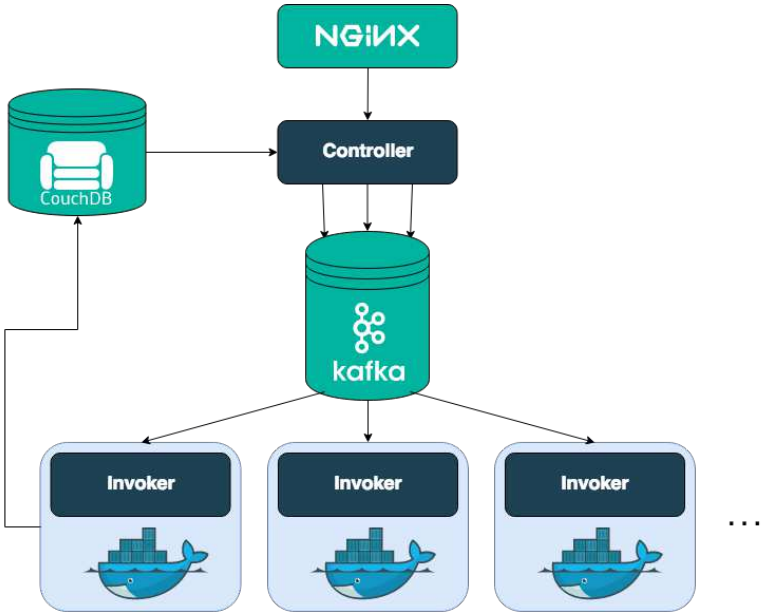


Figure 4.6: Architecture overview of Apache OpenWhisk [Apa19]

Kravchenko are part of IBM-Research and mention that "*installing a full blown Apache OpenWhisk would require about 2.5 GB of RAM (just the core components, before running any actions)*" [DB18]. The resource requirements make it tough to install OpenWhisk on a local machine. Running an unmodified version of OpenWhisk (that requires about 2.5GB of RAM) on an edge device like the Raspberry Pi (with only 1 GB of RAM) is clearly unthinkable and therefore OpenWhisk is not suitable for edge computing.

In order for Apache OpenWhisk to become usable on resource-constrained edge devices, Kravchenko and colleges tried to strip the architecture of OpenWhisk down to only the necessary parts [Kra17b]. They created multiple prototypes that aimed to allow OpenWhisk functions to work without a full OpenWhisk instance.

Apache OpenWhisk LEON (**L**ocal **E**xecution of **O**penWhisk actions with **N**odeRed) was an approach for open source serverless edge computing from Alex Glikson and Pavel Kravchenko [Gli17, Kra17a]. The aim was to be able to run OpenWhisk actions outside of a centralized cloud while keeping the actions compatible with cloud runtimes. LEON is marked as deprecated in Github since July 2017 [Kra17a]. In the meantime, the researchers worked on a similar project called Apache OpenWhisk Light.

Apache OpenWhisk Light is the successor of LEON. It is, as LEON, a research prototype based on OpenWhisk, Node-RED and Docker. Node-RED is a graphical tool that can be used on local edge devices like the Raspberry Pi. It allows to create data flows between triggers, Node-Red functions and Node-RED extensions. Apache OpenWhisk Light is implemented as a Node-RED extension that allows the use of Node-RED functions

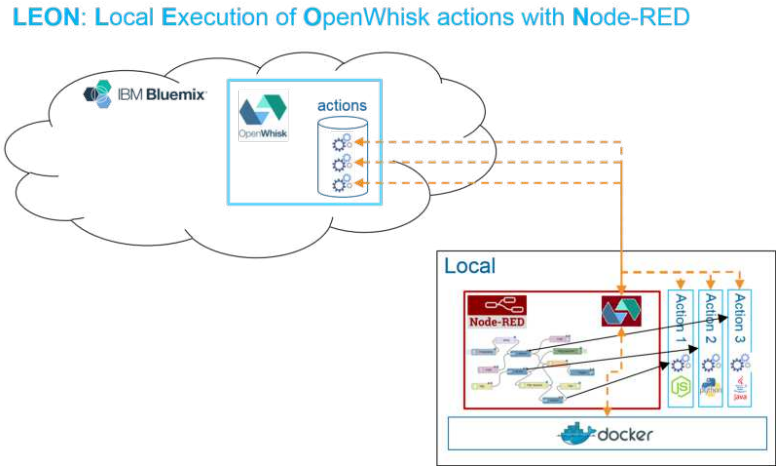


Figure 4.7: LEON: Local Execution of OpenWhisk actions with Docker and Node-RED [Gli17]

together with OpenWhisk actions. Figure 4.7 shows the basic concepts of LEON and OpenWhisk Light.

Due to lacking recent commits, the project seem to be abandoned in Github too. This is conflicting with the *Maintainer Requirement*.

Beside lacking maintenance, there is no documentation of supported hardware, the possibility of accessing machine learning accelerators (breaks *Hardware Requirement*) or model synchronization between edge and cloud states (breaks *Model Synchronization Requirement*).

Kravchenko and Glikson also demonstrate OpenWhisk Light's edge analytics capabilities. They use the image processing library OpenCV inside an OpenWhisk action (function) that is deployed on an edge device [Gli17]. The OpenCV based, serverless OpenWhisk edge device reduces bandwidth by cropping input images to smaller rectangles that only contain faces that were on the initial image and. Then it only sends those rectangles to a cloud service for image recognition, not the whole image. Beside OpenCV it is thinkable that other libraries, for example Sklearn, can be used with OpenWhisk Light. It therefore fulfills the Inferencing Requirement.

Cloud based Apache OpenWhisk supports custom runtimes based on Docker images [Web19c] and uses per default some interaction paradigms of the IoT (MQTT and HTTP) and therefore fulfills the *IoT Requirement* and the *Runtime Flexibility Requirement* [IBM19] too. Furthermore, there is good support for Python and some tutorials for machine learning inferencing on how to integrate popular ML-libraries like TensorFlow with the cloud based Apache OpenWhisk exist [(IB17]. This fulfills the Inferencing Requirement even better but still not the *Training Requirement*.

While OpenWhisk is a very valuable tool for serverless computing, its computational

complexity and the lacking documentation make it an unlikely candidate for practical serverless edge analytics.

Lambda@Edge

Lambda@Edge is part of the Amazon CloudFront CDN. Lambda@Edge allows running Lambda functions in response to events generated by the Amazon CloudFront CDN at the Edge CDN Nodes of AWS itself. It is primarily advertised as a programmable CDN that can dynamically serve different content for different users or devices based on the request body [AWS19l]

Figure 4.8 and Figure 4.9 show examples of Lambda@Edge in use.

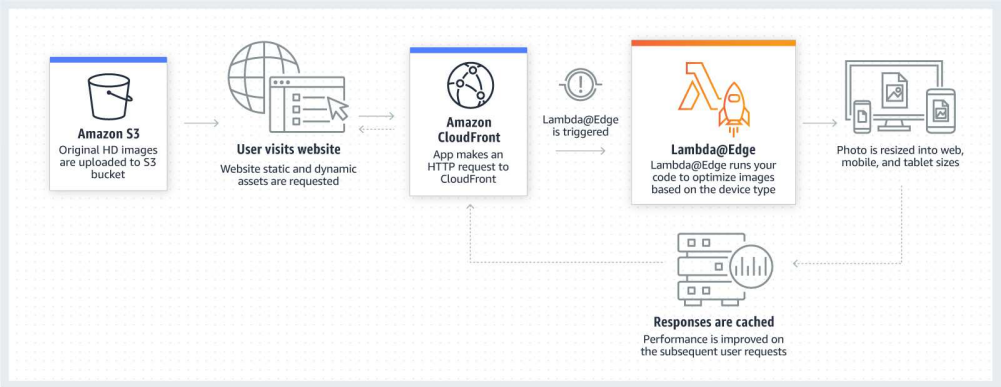


Figure 4.8: Lambda@Edge used to resize images at the CDN to fit the screen size of the user. Responses from Lambda@Edge are cached to further improve the performance [AWS19l].

Typical use cases of Lambda@Edge include various forms of user and device differentiation or localization during content delivery [AWS19l] to deliver the right amount or type of data for each device. Custom Lambda logic can be used for these differentiations. This improves the user experience by reducing the loading times and bandwidth needed for the end user.

AWS CloudFront only runs in various AWS data centers all over the world, no where else. So this framework is similar to edge computing but bound to local AWS' data centers. Other frameworks can be run on third party devices like, Raspberry Pi, too and are therefore more flexible.

AWS@Edge has stricter resource limits than the cloud based AWS Lambda (for Details about AWS Lambda see Section 2.2.5). While AWS Lambda functions are limited to a time out of 15 minutes until processing must terminate [AWS19k], Lambda@Edge limits the execution time to 5 seconds [AWS19a]. The function code of AWS Lambda functions is usually deployed as a compressed archive containing the handler code and all required dependencies. AWS Lambda limits the size of these archives to 50MB

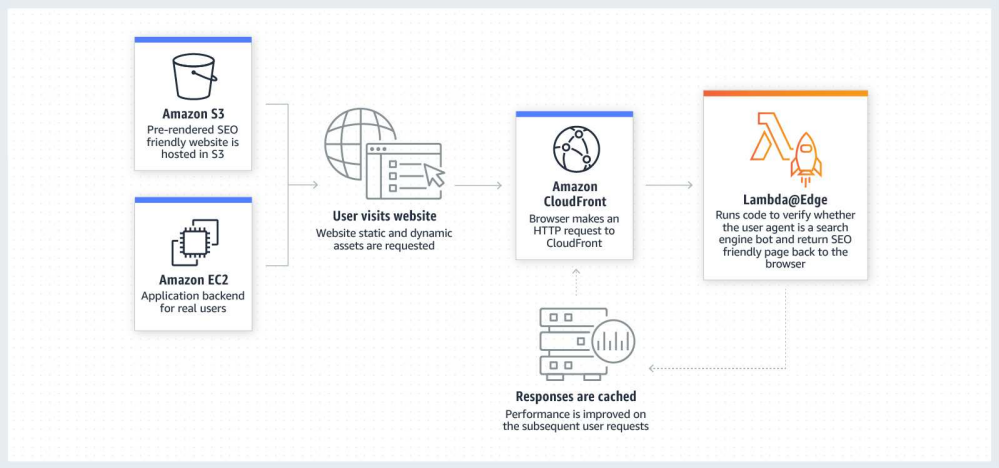


Figure 4.9: Lambda@Edge is used to decide whether the user agent is a search bot or a real user. Based on the decision an optimized website will be delivered [AWS19l].

[AWS19k]. Lambda@Edge limits the compressed code size of a Lambda archive to only 1 MB [AWS19a]!

Due to these extreme restrictions, Lambda at Edge is limited for edge analytics. The Execution Requirement states that machine learning functions require big deployment packages and long execution times. Machine learning training functions require many libraries and the dependencies currently don't fit into a 1 MB archive file. Specialized ML-hardware that is currently not available for Lambda@Edge (also not for AWS Lambda) would be beneficial.

The CDN data centers of AWS are limited to CloudFront events only, reacting to other events (e.g. for batch training, non- HTTP protocols like MQTT) is not possible there, the IoT Requirement is also not fulfilled.

Fortunately, AWS announced AWS Greengrass as a dedicated edge computing framework in 2017 [Ser17] that is less computationally limited and more flexible in terms of deployments and hardware. It runs on any device and also outside of AWS data centers making it more qualified for a heterogeneous serverless edge analytics framework. AWS Greengrass is discussed in Section 4.1.3.

Microsoft IoT Edge Functions

Microsoft IoT Edge was released into public preview in November, 2017 [Mic17a]. It extends Microsoft's Cloud (Azure) to edge devices and according to Microsoft, allows to deploy "*Azure Cognitive Services, Machine Learning, Stream Analytics, and Functions — and run them locally on devices from a Raspberry Pi to an industrial gateway using Azure IoT Edge*" [Mic17b].

The components of Azure IoT Edge are implemented as containers and need a container engine for startup. While the recommended container engine is Moby Engine, Microsoft mentions that Docker is also suitable because of it is very similar to Moby Engine [Mic17c]. Microsoft IoT Edge supports both Linux and Windows as a platform, but Linux containers can only be started on Linux hosts and Windows containers on Windows hosts [Mic17c].

In December 2018, Azure functions (most recent version: 2.0) officially support the languages: C#, JavaScript, Java and F#, in preview also Python 3.6 and TypeScript [Mic18d].

As already stated by the Community Requirement, Python is the preferred language of data scientists. However, for Azure functions, Python is currently only in preview and not production ready [Mic18a]. Using Azure Functions outside of the cloud, meaning on the Azure IoT Edge Gateway, is currently limited to C# functions, which breaks the Community Requirement.

Azure IoT Edge allows incoming connections via MQTT and Advanced Message Queuing Protocol (AMQP), communication via HTTP or any other protocols is not possible. As mentioned in the requirement analysis, MQTT is commonly used in the Internet of Things and Azure IoT Edge is, as the name suggests, usable for many use cases there!

Microsoft IoT Edge has officially supports Raspbian Stretch in its ARM32-version (the default Operating System (OS) of the Raspberry Pi) as well as other popular operating systems including Ubuntu AMD64-version, CentOS 7.5 ARM32 and AMD64 versions, Debian 8 and Debian 9 ARM32 and AMD64 versions [Mic18c]. This allows the framework, as required by the Edge Requirement, to be used on heterogeneous devices.

While Azure IoT Edge can run on various operating systems and devices, it currently lacks support for ARM64 devices. That is unfortunate because as recent edge computing devices, like the NVIDIA Jetson TX2, are based on an ARM64 architecture. Nvidia Jetson TX2 allows machine learning tools to run with accelerated performance with an energy efficient footprint, see Section 4.4.4 for details. While Azure IoT Edge doesn't officially work on Nvidia Jetson TX2, its opponent, AWS Greengrass officially supports these devices [AWS18b]. The Hardware Requirement is broken as Azure IoT Edge has no access to such machine learning accelerating devices.

Azure IoT Edge is very flexible regarding its execution model. The framework does not only allow deployment of serverless functions, it is made up of three main components: IoT Edge modules, IoT Edge runtime and a cloud based interface. The modules contain custom code and are created in the cloud - technically are they just simple Moby/Docker containers. During deployment, the containers are pushed from the cloud to the device. When developers want to deploy serverless functions on the edge device, they are packaged into a module (a container) and are afterwards deployed to Azure Edge within this container.

In 2017, only C# Functions were supported during the IoT Edge preview and functions

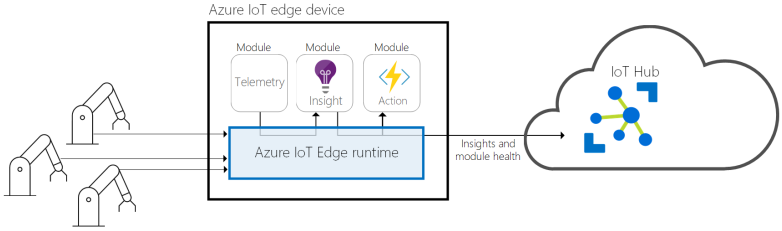


Figure 4.10: The Azure IoT Edge runtime [Mic18g]

could not run on ARM based Linux devices. In November 2018, Azure Functions 2.0 compatibility was announced for Azure IoT Edge but the supported programming languages are still restricted to C# only. [Azu18] While this already is a big limitation for serverless edge analytics, Azure IoT Edge also does not support more than 20 running modules per device [Azu19]. Being limited to only 20 unique serverless function modules seems to limit the idea of serverless, where servers should support as many functions as possbile in a very efficient way, for some users[3]. Therefore, IoT Azure Edge is too limited in its Execution Flexibility to be a suitable serverless edge analytics base.

For edge analytics (not serverless edge analytics) Azure IoT Edge is still a valid tool. The Inference Requirement is fulfilled by the built-in inferencing framework toolkit for machine learning models in Azure IoT Edge [Mic17d]. It works similarly to the mentioned deployment of Azure Functions. Model inferencing and its required dependencies are packaged into a Docker container. These container modules are first built in the cloud and then deployed to the edge devices. The machine learning modules are uncoupled from the serverless modules and a separate feature of Azure IoT Edge. Figure 4.10. illustrates the architecture of Azure IoT Edge in order to show that the ML modules and serverless modules are both just containers that are independent of each other [Mic18g].

While Azure IoT Edge supports machine learning models for inferencing, there is no official support for machine learning training. Due to the missing hardware support for GPUs or FPGAs and the restriction to only C# based functions, the use for machine learning tools in serverless functions is limited. While there is a whole research group at Microsoft Research that deals with Edge Machine Learning [Res19], they state that their focus lies on three approaches: attempting to compress existing large and accurate models to fit small devices [Res19] , "*creating new predictor classes that are specially designed for resource-constrained environments*" [Res19] from bottom up and on "*algorithms that dynamically decide when to invoke the intelligence in the cloud and how to arbitrate between predictions and inferences made in the cloud and those made on the device*" [Res19]. To name an example: One of their publications deals with the deployment of machine learning models "*trained on a laptop/other cloud [...] and shipped to an Arduino micro controller operating at 16MHz with 2 KB RAM and 32 KB read-only flash memory*" [KGV17]. That is much, much more resource constrained than hardware platforms like

[3]https://feedback.azure.com/forums/907045-azure-iot-edge/suggestions/33583447-iot-edge-module-deployment-limit Accessed 16.03.2019

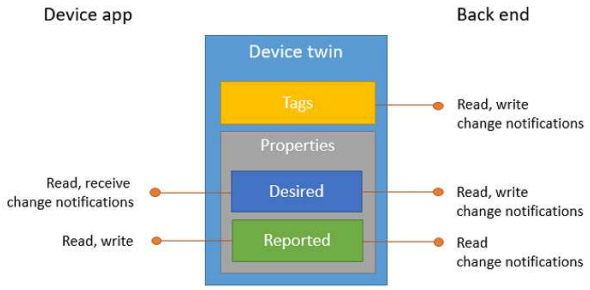


Figure 4.11: Azure Device Twins Concept [Mic18e]

Raspberry Pi's. So after all, while Microsoft is working on edge machine learning, it is mostly edge inference based and specifically for Azure IoT Edge, there are no machine learning training resources available. The Training Requirement is not satisfied.

The state synchronization requirement can be fulfilled partly with Azure IoT Edge's internal tools. Similar to AWS Greengrass, Azure IoT Edge offers a state management system for IoT devices and edge modules [Mic18e, Mic18f]. Microsoft calls these concepts Device Twins and Module Twins. A twin is represented by a JSON document that represents the state of the device or module. The document is synced between the Azure IoT Hub cloud instance, the local module or device itself and if available an Azure IoT Edge instance [4]. Figures 4.11 and 4.12 illustrate that module and device twins can be accessed by the cloud backend and the devices themselves. The JSON documents are used to describe a device or module. Each IoT device and each edge module connecting to Azure can have one device or module twin. The synced twins are used to give access to device states, even if those devices are not connected to the system anymore.

Each JSON consists of two parts: Tags and Properties. Tags can only be accessed by the cloud backend only and are usually used to store information describing the device or module. Properties can be separated into desired properties and reported properties. Desired properties can be read and written by the cloud backend and are read-only by the IoT device or module. Reported properties work the other way round, they can be read and written on the devices or modules and are read-only by the cloud backend. The IoT devices or modules can be informed about property updates while the cloud backend can only query them [Mic18e].

The size of these device and module twins is limited to "*an 8KB size limitation on each of the respective total values of tags, properties/desired, and properties/reported*" [Mic18e]. According to an Azure IoT Team Admin in 2018, Microsoft plans to increase the size limit but can not give any disclose date yet [5]. 8KB is very, very limited and using Twins

[4] https://github.com/MicrosoftDocs/azure-docs/issues/18767

[5] https://feedback.azure.com/forums/907045-azure-iot-edge/suggestions/33583492-iot-hub-device-and-module-twins-limit

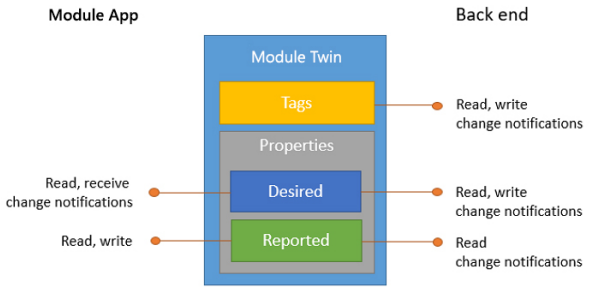


Figure 4.12: Azure Module Twins Concept [Mic18f]

to synchronize machine learning models seems to be unreasonable due to this. Accoring to Ishakian et al. ML-models can easily reach up to 500MB and more [IMS17].

Using module and device twins for syncing very basic machine learning models that could be created on edge devices might still be sufficient. More details are given in the implementation chapter and in the further path of this thesis.

AWS Greengrass

AWS was the first cloud provider who published a Serverless Cloud Computing platform with AWS Lambda in 2014 [AWS18c]. Since then, their framework has been used for many research and business projects [AWS18c]. Furthermore, AWS Lambda was recently extended to also work on Edge devices. Branded as AWS Greengrass AWS is extending their Lambda framework to the edge. Greengrass enables local edge devices to handle triggers and run AWS Lambda functions locally. Furthermore, it handles the configuration and runtime environments, deployments from the cloud and MQTT communication for the AWS Lambda functions.

AWS Greengrass is similar to Microsoft's Azure IoT Edge framework but in some ways still very different. While both frameworks allow deployments of serverless functions on edge devices, AWS Greengrass only allows the deployment of serverless functions. Microsoft IoT Edge also allows the deployment of any continuous-running, arbitrary docker containers.

AWS Greengrass has support for AWS Lambda functions and comes with additional features including a local2cloud messaging gateway for forwarding, cloud synchronization of device state synchronization with device shadows (similar to Azure device twins) and a package called Greengrass ML that is capable of running ML-inference - in means of a ML-model that was trained in the cloud and can automatically be deployed to be used for predictions, not for training - on the serverless edge devices by AWS Lambda functions. Thats similar to edge based MLaaS.

AWS Greengrass fulfills most of the Requirements from Section 4.1.2.

As a dedicated edge computing serverless platform it fulfills the Edge requirement.

Some required interaction paradigms of the IoT are also supported. AWS Greengrass has a build-in MQTT server and a MQTT-cloud-bridge [AWS19j]. Beside MQTT, AWS Greengrass supports OPC-UA, an information exchange standard for industrial communication [AWS19f] and since Lambda functions running on Greengrass have access to network resources, according to AWS, support for any protocol that rides on top of TCP-IP can be implemented [AWS19f][6]

The languages officially supported by AWS Greengrass, in November 2018, are Python 2.7, Node.JS 6.10, Java 8, C and C++ [AWS18c]. Python is a commonly used language of data scientists and is required by the Community Requirement. According to the study about the Data Science Community, R is another language that often used for statistical computing and machine learning. Bryan Liston has shown for the AWS Compute Blog that it is also possible to compile and execute code written in R (with the help of some wrapper code) in an AWS Lambda function [Lis16]. Oracles FastR executes R code on the JVM and can therefore also be used to run analytic R code in Lambda. If this also works on Greengrass devices and the AWS Lambda functions on Greengrass is still unknown and has yet to be shown.

Greengrass is actively maintained by AWS. In March 2019, Greengrass version v1.8 was released and it constantly gets updated, therefore, the Maintenance Requirement is fulfilled.

The execution containers of AWS Lambda are very flexible but still limited. AWS Greengrass supports traditional serverless functions that get started up due to certain triggers and supports long-running functions. Long running functions will not stop running but still need an initial trigger to be started. For long-running functons, no timeouts will stop the function. Those long-running functions are very suitable for training larger machine learning models where the regular timeout of AWS Lambda, which is 15 minutes, is too short. Beside long-running functions, AWS Lambda has recently got support for custom runtimes. Greengrass does unfortunately not yet support this. Lambda functions deployed on Greengrass are not more restricted than in the cloud. The size of the function (a zip archive) is still limited. Though, with 50MB compressed size, it is big enough for some python libraries. While some libraries will require more space, many could be sized down further by removing unnecessary code like comments and inactive code.

Regarding machine learning, AWS Greengrass comes with a built in inferencing engine, Greengrass ML, based on AWS Sagemaker, that supports the open source machine learning frameworks TensorFlow and MXNet [Gre18]. Machine learning models can be trained in the cloud and be used for inferencing on the edge. Currently, it is only possible to access such inferencing models, when they are deployed on a Greengrass device, with AWS Lambda functions that are based on Python. This fulfills the Inferencing requirement.

[6] Found in footnote: “Next Steps”

Due to official support for NVIDIA Jetson TX2, the Hardware Requirement is also fulfilled. AWS Greengrass uses CUDA for its inferencing libraries based on the TensorFlow v1.10.1 and MXNet v1.2.1 binaries. Beside NVIDIA Jetson TX2, there is support for the Raspberry Pi 3 (its also their default edge device) and devices running on Arch Linux (ARMv8 only), Amazon Linux/Ubuntu 14.04-16.04 (x86_64) and Raspbian Stretch (ARMv7l). This wide range of supported platforms makes Greengrass very flexible and enables deployments on heterogeneous devices. It is also worth mentioning, that if Greengrass is not required to run natively, there is the possibility to run AWS Greengrass in a Docker container on Windows, macOS and Linux platforms [AWS19d].

While inferencing and hardware support are very good, Greengrass has no integrated features for machine learning training and also no support for federated learning. The Training Requirement is not fulfilled but still not completely broken. Greengrass supports machine learning hardware, has a very flexible execution environment and support for Python. Furthermore, Greengrass' included features for state synchronization (AWS IoT Device Shadows) could possibly be used to synchronize machine learning models between edge devices.

The Synchronization Requirement is also partly fulfilled by Greengrass. As already mentioned, AWS Greengrass' integrated state synchronization model, IoT Device Shadows, could be used to synchronize machine learning parameters with the parameter servers. AWS IoT Device Shadows only support a maximum payload of 8KB and are quite limited because of that. But Greengrass does also support communication with the AWS IoT Cloud via an MQTT uplink that gets synchronized after an outage as soon as the Greengrass bridge goes online.

4.1.4 Comparison and Conclusion

In conclusion, after analyzing the mayor serverless edge computing frameworks, AWS Greengrass can be considered as the most mature framework today. Table 4.1 shows which requirements for serverless edge analytics are satisfied by the frameworks. AWS Greengrass clearly fulfills most of the requirements and is therefore a valid base for additional research and a good base for the prototype of this thesis.

To answer the first research question (RQ1.1), "*What are suitable frameworks for serverless edge analytics?*" the relevant frameworks were discussed. Flogo, OpenWhisk Light, Azure IoT Edge, AWS Greengrass are all suitable for serverless edge computing. With support for Machine Learning inferencing, these frameworks are also suitable for some serverless edge analytics projects that do not require machine learning training.

Answering the second part of first research question: the analysis clearly shows that most platforms lack support for common analytic frameworks and machine learning tools implemented in R and Python, lack ML hardware support, flexible execution containers and long running functions. Some support inferencing but do not support the training of machine learning models and lack built in features to synchronize insights between the edge nodes. "*Which parts of the suitable frameworks need to be used and which modifications*

are required to support machine learning on these platforms?" AWS Greengrass turned out to be suitable for a serverless edge analytic training prototype because of its flexibility and platform support. Modifications regarding the built-in support support of local ML libraries, federated learning, proper function monitoring and logging as well as state synchronization with other devices is still missing. Furthermore, the research below will give details on Greengrass monitoring capabilities and that they are not sufficient and need improvement.

The next sections are based on AWS Greengrass because it is selected as the most suitable serverless edge computing framework throughout this analysis.

	FLO	OW	OWL	MIE	AGG
Edge Requirement	Yes	No	Yes	Yes	Yes
IoT Requirement	Yes	Yes	Yes	Yes	Yes
Community Requirement	No	Yes	Yes	No	Yes
Inference Requirement	Yes	No	No	Yes	Yes
Training Requirement	No	No	No	No	Partly
Maintainer Requirement	Yes	Yes	No	Yes	Yes
Hardware Requirement	Partly	No	No	No	Yes
Execution Requirement	No	Yes	Yes	No	Yes
Synchronization Requirement	No	No	No	Partly	Partly

Table 4.1: Summary of the fulfilled requirements for each framework. Flogo (FLO), OW (Apache OpenWhisk), OWL (Apache OpenWhisk Light), MIE (Microsoft Azue IoT Edge), AGG (AWS Greengrass)

4.2 RQ 1.2: Metrics in Serverless Edge Computing

This section covers research question 1.2: "*What are means and suitable metrics for monitoring the performance and Quality of Service (QoS) of a serverless edge computing function? How does this differ from ordinary serverless functions?*"

4.2.1 Methodology

In order to find metrics that are suitable for measuring the performance and service quality of serverless functions, a literature analysis was carried out in Section 4.2.2. The scope of the analysis are metrics in cloud computing, metrics in edge computing and metrics in the context of serverless functions and the definition of performance metrics.

Section 4.2.3 discusses the differences between metrics for cloud based versus edge based serverless functions. It is based on the knowledge obtained form Related Work analysis in Section 3.

Section 4.2.4 provides details on the metrics that are assumed to be most suitable for measuring the performance and quality of a serverless edge computing function. These assumptions are derived from Section 4.2.2 and Section 4.2.3.

4.2.2 Metrics Literature Analysis

Metrics are existent in many different areas of cloud computing. To separate those, the following three paragraphs distinguish between metrics in Cloud Computing, Edge Computing and Serverless Computing.

Cloud Computing The intense use of Cloud Computing raised research awareness to Service Level Agreements (SLA) between cloud providers and cloud computing customers [ADC10]. The expectable quality and performance metrics of cloud services as well as their negotiated values are set by the SLAs. The results of the cloud services need to be comparable and the metrics enable that. Alhamad et al. listed various SLA metrics in 2010 [ADC10]. They are intended to make cloud providers comparable before acquiring server capacity from them. Table 4.2 shows some selected metrics that are useful for comparing cloud service offers of different providers:

Alhamad et al. also state that the different types of Cloud computing (see NIST Types in Section 2) focus on different kind of metrics. IaaS traditionally focuses on usage metrics like usage time, CPU and memory usage, overall response time, ability to scale and the service availability. Any metrics regarding billing and operational performance analysis are of particular interest here.

Users of SaaS platforms are usually not interested in hardware related metrics like CPU or memory. SaaS users prefer to focus on metrics that are valuable for their business, their use cases and on specific runtime metrics of the used SaaS software. Alhamad et al. name geographic location, privacy restrictions (that might be depended on a certain

CPU capacity
Memory Size
Boot time
Storage
Scale Up
Scale Down
Availability
Response time
Integration
Environments of deployment
Geographic location
Privacy
Transferring Bandwidth

Table 4.2: Selected cloud computing metrics [ADC10]

geographic region) and transferring bandwidth as a typical example for a Storage as a Service platform (which is a Software as a Service platform itself) [ADC10]. Examples that are interesting for SaaS users include the duration that is needed to create new users, duration for adding a ticket in a ticket tracking system, average response time of the application, availability and operating system compatibility.

The metrics listed above can generally be categorized into two categories: static and dynamic metrics. Static metrics are properties of software that do not change over time and might only be used for comparisons of the software with other software , as Alhamad et al. did in [ADC10]. For instance, the number of supported browsers is a good metric for comparing the suitability of competing SaaS services and can be considered as static. For example: Software with more supported browsers can be interpreted as a more flexible software - this metric is relatively static as it will not change often. Another example of a static metric is the maximum number of concurrent requests that can be executed per user account - it will not change often.

Dynamic metrics are used to compare the performance of a service, and not for comparisons of static properties of a service. Dynamic metrics change over time, a good example thereof is the CPU usage (percentage of used CPU resources) that is regularly changing. Dynamic metrics change continuously and help measuring and comparing the current performance of a system. The number of CPU cores and corresponding frequencies can be considered a static metric, the used percentage of those cores is a dynamic metric.

Edge Computing The main purpose of edge computing is the reduction of the total used bandwidth and response time [SCZ+16]. Therefore, users of edge computing systems are interested in metrics regarding the total network usage and network latency.

Shi et al. discuss four general metrics in edge computing [SCZ+16]. They are listed in Table 4.3 and are discussed below.

Latency
Bandwidth
Energy Use
Costs

Table 4.3: Common metrics in Edge Computing [SCZ+16]

Latency: Shi et al. explain that latency is one of the most important metrics to evaluate the performance of edge computing systems. Latency depends on the computation time of each computation step and on the network delays between the requester and the computational resources [SCZ+16]. According to them it is not sufficient to use the nearest edge device for computation in order to achieve the lowest latency. Finding the optimal device between the user and the cloud, in terms of the computation latency that is generated by the edge or cloud node plus the network latency to reach the node, optimizes performance. Latency is usually measured in milliseconds and either be defined as one-way latency (time from sending a packet at the source to receiving the packet at destination) or as round-trip (similar to end-to-end) latency (time from sending a packet at the source to receiving an answer packet from the destination back at the source) [KZA99].

Bandwidth: Another performance metric listed as important in Edge Computing systems is used bandwidth [SCZ+16]. High transmission rates, especially for big files, require high bandwidth. Transmission reliability and possible bandwidth is enhanced when the transmission path is shortened [SCZ+16]. The total required bandwidth is an aggregated metric that sums the dynamic metric of currently used bandwidth between each node of the system. For example when a sensor sends 5 Megabyte (MB) to the edge gateway and the gateway forwards the 5MB to the cloud, a total of 10MB were transmitted. Offloading edge computation to cloud servers or other edge devices may reduce the load of overloaded edge gateways but increases the total used bandwidth of the system. In terms of bandwidth, it would therefore always make sense to at least preprocess data at the nearest edge nodes and to reduce the (for example) 5MB to 2MB before sending the data to the cloud. As mentioned in the Related Work Section, Nastic et al. propose Serverless Realtime Edge Analytics especially for such use cases [NRS+17].

Cloud providers usually bill for used bandwidth for incoming and outgoing connections in MB. Connections that stay inside of the data center (e.g. between two service instances) are usually free. The costs for the used bandwidth can quickly add up! AWS takes between $0.05 and $0.09 per Gigabyte (GB) for connections from EC2 to the Internet but charges nothing ($0.00 per GB) for incoming connections from the Internet to AWS [AWS19c].

Cloud data centers are highly specialized environments. Edge computing systems are

usually bound to lower bandwidth transmission rates and lower processing power. This is due to the unmanaged and unreliable environment that edge computing systems operate in. Bandwidth costs are often much higher for edge computing systems than in traditional cloud environments. To mention an example: In 2019 in Austria, a high speed LTE contract for 10GB monthly costs around 6€[7] which is 10 times the cost of the AWS cloud. LTE contracts with unlimited bandwidth exist too but those are often still limited, intransparently, due to fair-use policies. It becomes clear that the used bandwidth is a huge cost factor for distributed systems.

Energy Use: "*Battery is the most precious resource for things at the edge of the network*" [SCZ+16] and therefore the energy tradeoff between computing locally and offloading the workload (or parts of it) to the cloud is critical. According to Shi et al. network signal strength, data size and available bandwidth influence the transmission energy overhead. If a system is optimizing itself based on energy usage, network offloading is only worthwhile if the energy use of local computation is higher than offloading. Of course, a platform needs to be configured to optimize along certain metrics because energy efficiency will often conflict with low latency or low costs. Local computation might be more latency efficient but less energy efficient.

Costs: Beside latency, bandwidth and energy use, the cost metric is also very relevant in edge computing. Service providers often optimize their processes based on costs. Costs are often conflicting or synergizing with the other metrics. In general, optimizing an edge computing system along specific metrics can result in conflicts with other metrics [NRS+17]. Reduced bandwidth or reduced energy consumption can on the one hand reduce the operational costs but on the other hand can, depending on the environment the system operates in, also increase the costs. Therefore, it is important to specify the metrics that influence the configuration of an edge computing system first.

Serverless Computing McGrath and Brenner implemented an early serverless computing platform based on the .NET framework [MB17] in 2017. As part of their work they also focused on measuring the performance of the executed functions. They are listing measures that impact the execution quality for the function runs (see Table 4.4).

Apart from traditional quality measures of distributed systems (including network latency and bandwidth), they mention "*variations between language runtimes and function code size, system-wide performance of serverless platforms, performance differences between event types, and CPU allocation scaling*" [MB17] as possible implications of the performance of function executions (see Table 4.4). This implies that the performance of serverless functions should not only be measured based on hardware and network utilization metrics. Measurements should also include the metrics that influence serverless function runs like runtimes, code size, event types, function RAM and corresponding CPU allocation limits (and associated scaling properties). Only this ensures that the performance of individual function runs can be properly compared to one another.

[7] https://www.spusu.at/spusu10gb

Runtimes and Language
Function code size
System Performance
Event types
CPU allocation scaling

Table 4.4: Common metrics in Serverless Computing [MB17]

AWS Lambda is the most popular serverless computing framework. In AWS Lambda, function runs can be monitored along various metrics that are persisted directly or are part of the functions log-file. Based on the runs and the corresponding logs from the functions, various metrics can be extracted [AWS19i]. Table 4.5 shows the metrics that are automatically collected on each invocation by AWS Lambda. The information is usually pushed to AWS CloudWatch on each invocation. AWS CloudWatch is a monitoring service by AWS, separated from AWS Lambda, and is the default monitoring service for many of their services (including AWS Lambda).

Total Invocations
Total Errors
DeadLetterErrors (Permission errors, Throttles from downstream services, Misconfigured resources, Timeouts)
Execution duration in milliseconds
Total throttles due to concurrency limits
Iterator Age
ConcurrentExecutions
UnreservedConcurrentExecutions

Table 4.5: CloudWatch Metrics of AWS Lambda [AWS19i]

While CloudWatch collects multiple metrics, AWS Lambda is also storing execution logs for each AWS Lambda invocation. There, additional runtime metrics can be extracted. For instance, the working memory actually used as well as the allocated (billed) memory are persisted in the execution log of the function.

4.2.3 Metrics on Edge devices vs Metrics in the Cloud

Collecting metrics of serverless functions is different on edge devices than it is in cloud environments. In Section 4.2.2 - where different metrics in the field of cloud computing, serverless computing and edge computing are listed - it became clear, that different types of cloud computing (e.g. Edge Computing, Serverless Computing, IaaS or SaaS) focus on different types of metrics (e.g. IaaS mostly focuses on hardware metrics while SaaS also focuses on software metrics).

In order to find suitable service providers, static metrics are preferred. But for comparing the performance of services during runtime the focus should be on dynamic runtime metrics.

As explained in Chapter 2, serverless computing can be considered as an own type of cloud computing.

Cloud based serverless platforms can usually be categorized as FaaS, meaning that customers pay for each function invocation. Only some serverless cloud frameworks can also be hosted on-premise by the customers themselves (e.g. Apache OpenWhisk). FaaS customers don't care about the underlying server hardware, as FaaS is serverless and provided as a service.

Due to their novelty, edge-based serverless frameworks (with the exception of AWS Lambda@Edge) are usually self-hosted and not provided as a service. They do not support the Function as a Service paradigm and customers must take care of the hardware that is hosting the serverless edge nodes themselves.

Serverless Edge Computing is therefore (based on today's tools) not really serverless in terms of reduced maintenance of underlying servers and pay-as-you-go business models.

To provide an overview, mayor Serverless edge computing frameworks are discussed in Section 4.1.3.

The fact that cloud-based serverless frameworks focus on the FaaS paradigm while edge-based serverless frameworks focus on simplifying edge computing services and deployments results in some contradictions regarding their runtime objectives and requires a different view on metric collection for the two approaches. Cloud-based FaaS focuses on runtime metrics regarding the execution quality of each function invocation, while edge based serverless computing focuses on the execution quality of the functions as well as on the runtime metrics of the serverless framework hosts.

Frameworks, like AWS Greengrass can only be installed on own machines (or VMs). AWS Greengrass is not provided as a service. AWS Lambda can also execute AWS Lambda functions but executions are provides as a service and in the cloud. The main advantage of cloud-based FaaS is the reduce maintenance workload for the underlying servers and pay-as-you go business models. Therefore, typical cloud-based FaaS customers do not like to manage, operate and monitor underlying hardware for their function runs. Customers of self-hosted serverless frameworks like Apache OpenWhisk or edge based self-hosted services like AWS Greengrass accept to run, monitor and maintain their own hardware and are therefore more focused on the monitoring capabilities of their systems.

Metrics of an edge based serverless function Due to the conclusion that AWS Greengrass is the most suitable candidate for serverless edge analytics as part of research question 1.1 [4.1.3], AWS Lambda and AWS Greengrass were chosen as the base line for the further discussion in the frame of this master thesis.

As serverless edge computing is different from ordinary FaaS, it is important to discuss the difference in metric collection between cloud and edge based serverless software:

A FaaS oriented view focuses on metrics that are relevant for the performance of each single function run and that are relevant for billing those invocations, cost savings and optimizing each function invocation. Beside these objectives, there are also other metrics that are important for a FaaS service. McGrath and Brenner mentioned that cloud based serverless functions are not only influenced by hardware performance but also by the software of the serverless computing provider (e.g. runtime, coldstart vs warmstart, etc.) [MB17]. Not only the traditional metrics like CPU and RAM utilization are valuable metrics for estimating the function performance but other FaaS metrics like software versions, cold-vs-warm starts or function size matter for monitoring too.

Edge based serverless functions do not (with some exceptions) run in a cloud-based FaaS environment. For AWS Greengrass, serverless edge functions run on edge nodes (servers), that are mostly handled by the same team as the functions that run on those nodes. Metric collection is therefore broader than the mentioned metrics for FaaS services. To use Edge-based serverless frameworks, infrastructure metrics as well as edge computing specific metrics need to be incorporated. Metrics that are useful for monitoring a serverless edge computing node include the hardware utilization metrics of the host (similar to IaaS metrics) in regular time intervals as well as the execution metrics of each serverless function invocation (like startup time, coldstart/warmstart, payload size).

Beside the difference in metric collection, the observability and monitoring capabilities of cloud servers and edge nodes are different. Cloud servers usually run as virtual resources on clusters of hardware. In edge computing, machines lack computational resources and are bound to energy limits. Therefore, they often run the serverless frameworks on their operating system natively, directly, without virtualization or with less virtualization. This results in an off-trade regarding the monitoring capabilities - to name an example: cloud based serverless computing frameworks like Apache OpenWhisk usually spin up Docker containers to execute the serverless functions. In such containers, the network interfaces and hardware use can easily be monitored. In edge computing, frameworks, like AWS Greengrass, often run their functions natively [8] [9]- AWS Greengrass was not containerized before version v1.7 and are still configurable to not be containerized - on the operating system and are therefore usually harder to monitor/observe. Section 4.3 and Section 3 provide more details on monitoring serverless edge computing functions e.g. as Das et al. by using top or dockerstats [DPW18].

[8] https://docs.aws.amazon.com/greengrass/latest/developerguide/lambda-group-config.html#lambda-containerization-groupsettings

[9] https://docs.aws.amazon.com/greengrass/latest/developerguide/run-gg-in-docker-container.html

4.2.4 Suitable Metrics for Serverless Edge Computing

Serverless edge analytics is a broad field where technologies from the fields of edge computing, serverless computing and analytic frameworks are merged together. Each of these fields value specific metrics. Due to the identified problem of this thesis - serverless edge devices being overloaded very often - the focus of this section is finding relevant metrics that can be used to predict the processing time of a serverless function on an edge device. Therefore the selection of only a subset of the entirety of collectible metrics, by focusing on the scope of this thesis based on the findings above, is required. This chapter will elaborate on the selected subset.

The following list shows the metrics that could potentially be selected to be used in the prototype for this thesis in Section 5. Each metric is briefly described and justified.

- **Latency**. Low latency is a critical factor for many applications in edge computing. Latency can be measured for any network connection but in edge computing it makes sense to focus on two targets: Measuring latency between the end user and the edge node as well as between the edge node and a cloud data center. Latency is usually measured in milliseconds and either defined as one-way latency (time from sending a packet at the source to receiving the packet at destination) or as round-trip latency (time from sending a packet at the source to receiving an answer packet from the destination back at the source) [KZA99]. Most applications focus on minimizing round-trip latency. Round-trip latency is composed of the one-way latency from the source to the target plus the computational delay at the target plus another one-way latency from the target to the source. The computational latency depends on the payload that is sent to the target and on the computational resources that are available at the target. By nature, Edge computing should already reduce the latency of serverless functions as the network distance between the user (event source) and the server is reduced. Indeed, that is not always the case. Baresi et al. found, that Edge computing can drastically increase system performance as long as the edge device is not overloaded [BMG17]. As soon as overloading happens, processing time and latency increase into unacceptable sizes [BMG17].
- **Bandwidth**. As mentioned in Section 4.2.2, the total consumed bandwidth is an important metric in edge computing [SCZ+16]. Sending and receiving data from and to the cloud results in high bandwidth costs. Reducing the bandwidth needs is not only necessary for the implementation of a serverless edge analytic tool but is also important for decisions during runtime. Especially when considering forwarding payload to the cloud, the used bandwidth can be the most influential factor regarding the costs (because bandwidth is often very expensive) but bandwidth can also influence the latency of the forwarding.
- **Computational Load**. Computational load can be measured through various metrics. Most commonly CPU, RAM and network utilization are used. Those

metrics can easily be compared with one another and clearly show whether a device is idle or overloaded. Collecting these metrics in serverless edge computing can happen in two stages. Firstly, the utilization of the whole edge node can be collected at each function invocation event. Secondly, the utilization of each function could furthermore be measured. As Section 4.3 shows, monitoring the exact resource use of AWS Greengrass functions is not feasible but by the help of some monitoring frameworks that potentially allow the exact monitoring of each single function run (similar to how these frameworks work for FaaS runs in cloud environments, eg. like AWS Lambda) could solve some problems.

- **Analytic Metrics**. The analysis on available edge devices and frameworks in Section 4.1.3 has shown that some frameworks support machine learning accelerating devices like GPUs. Serverless functions (especially serverless analytic functions) that are able to use these resources benefit from a performance boost. Comparing metrics of computational load (eg. CPU and RAM) to estimate the execution time of a serverless function is only valid as long as the metrics were collected on similar environments. If a device has no access to machine learning accelerators, the CPU will be used and CPU utilization will naturally rise. It can therefore be necessary to distinguish between measurements on devices with enabled or disabled machine learning accelerators. Due to the novelty of edge devices with integrated accelerators and the variety of such solutions, there is no common utilization metric for these devices. Therefore, the pure availability of such accelerators could for example be collected as a Boolean metric.

- **Payload Metrics**. Each time a function gets triggered, the invocation comes with some payload that can be important for predicting the latency. This can range from the size of the payload, the invocation time as well as many other metrics and also very specifically for edge computing the location of the request. To give an example, during a football match, the edge nodes in a football stadium might be overloaded. Overloaded nodes happen at a specific time and location. Predicting such events, based on payload like for instance timestamps or specific file sizes, can help to prevent overloading by rerouting requests as necessary.

It is not clear which exact metrics will be necessary to train an optimal model for reducing the latency of serverless function invocations by placing the computational part of the function invocation on the edge or cloud. However, thanks to the analysis above it is clear that some metrics are particularly important, more so than others. It is not part of this thesis to find the optimal metrics for a use cases, instead, a general discussion is desired.

In the list above, specific metrics, such as energy use, costs, disk utilization, were purposefully excluded from the scope of this thesis as they do not or are not expected to influence the execution time of a serverless function significantly for first simulation attempts. Others were left out as reacquiring the data is incredibly challenging and above

the scope of a master's thesis. However, adding more metrics to the study has potential for further research projects. To name an example: monitoring the energy use on an edge node (like e.g. a Raspberry Pi 3) is very difficult and may even require additional hardware and is out of scope for this thesis.

4.3 RQ 1.3: Monitoring and Storage

In this section, research question 1.3 (RQ1.3: "*How can serverless edge computing devices and the corresponding function runs be monitored and stored (e.g. in AWS Greengrass)? Which tools are required?*" 1.3) is studied.

4.3.1 Methodology

In order to find suitable monitoring approaches for AWS Greengrass, the methodology of this chapter consists of an online research about monitoring, existing monitoring tools for AWS Lambda and monitoring tools for AWS Greengrass. Furthermore, the installation files of AWS Greengrass itself were analyzed in order to understand the extent of information that can be gained from the Greengrass software itself. The file analysis was necessary because, as the online research has shown, AWS Greengrass lacks proper built-in monitoring capabilities. By discussing the structure of AWS Greengrass, places where custom monitoring interfaces can be added to the framework as well as license restrictions will be shown.

4.3.2 Monitoring Introduction

Monitoring is the task of capturing and persisting metrics of services, components and infrastructure [MS04]. It is usually done to keep functional and non-functional properties of running services accessible over time.

Monitoring can be done by different stakeholders in a system, for instance, end users or platform providers. This makes it easier to separate monitoring into several perspectives from which a system is observed. Usually there are different responsible organizations or persons for each of the perspectives.

The following paragraphs describe the different perspectives that exist in traditional distributed systems, serverless cloud platforms and serverless edge platforms.

Traditional perspectives Traditionally, the monitoring of network services can be seen from two different perspectives:

Firstly, there is the perspective of the end-user, where monitoring happens on the machines and devices of end users. The remote service can be considered a black box and monitoring and benchmarking happens on the machines and devices of the user. The aim is to observe, whether criteria that are relevant and measurable for the users, are met.

Secondly, there is the perspective of the service providers themselves, which focuses on the private (internal) monitoring of the services to meet the criteria of the end users. For instance, this can include the performance of separate blocks of code, hardware utilization, availability, energy use and many other internal views.

Serverless perspectives While, theoretically, the monitoring of serverless platforms can also be separated into the two traditional perspectives, it's even more useful to separate monitoring of serverless functions into three perspectives.

The first one is again the end-user perspective. In fact, it is equal to the first traditional perspective because end-users of serverless functions treat such services similar to traditional services because their understanding of a remote service is a black box, with an unknown internal structure. End-users usually monitor performance by accuracy, latency, bandwidth and costs to check if their criteria are met.

The second monitoring perspective of serverless platforms is the view of developers. In serverless, developers may care more about monitoring their internal processes than in traditional services. Developers of serverless functions care about the exact performance of their functions because, as Yan et al. state, the runtime as well as the used resources of serverless functions are directly relevant for billing: "*Serverless abstracts away the notion of a server from the developer, and pushes operational concerns to the providers of the serverless runtime. Despite this, it will be important for developers to monitor the deployment of their solutions since the execution of their functions is directly mapped to a cost model that charges for execution time (typically) and not for idle time*" [YCCI16].

Monitoring the internal components of the serverless services themselves is relevant for performance measurements of the third perspective of serverless platform monitoring, the one of serverless platform operators themselves. Operators of serverless platforms are interested in monitoring all layers of their service. Apart from the used resources of the developer-monitored serverless functions, the platform operator is known to care about infrastructure costs, deployments performance, availability, security and invocation intervals.

Serverless edge perspective Monitoring of serverless edge computing functions is similar to the serverless platforms. The identified suitable serverless edge computing platforms from RQ1.1 in Section 4.1.3, AWS Greengrass and Microsoft Azure Edge, are not available as FaaS platforms. However, these serverless edge frameworks are available for developers as on-premise (self-hosted) only. This means that both the functions that are running on top of the edge nodes as well as the platforms themselves need to be monitored by the developers. While AWS still handles some parts of the whole AWS Greengrass ecosystem, e.g. the deployments of fresh function code, the installed instances on the edge devices are the responsibility of the developers.

In other words: the first, second and third perspective of monitoring still exist in serverless edge computing but the responsibilities of these perspectives have changed. In current serverless edge computing frameworks, the developers are responsible for the second and third monitoring perspectives of serverless computing.

The first perspective of serverless edge computing is, similar to the traditional and serverless perspectives, the perspective of the end-users.

The second perspective is the view of the function developers, where the individual function performance is important. The invocation of an AWS Lambda function running on an AWS Greengrass core is not billed by AWS, therefore it's not important to monitor the functions performance for the reason of billing. However, the overall system performance of each node heavily depends on the performance of each function and therefore the performance of each function still needs to be monitored in order to detect internal problems of functions.

The third monitoring perspective of serverless edge computing regards the stability of the edge node themselves and is similar to the platform monitoring in regular serverless computing. The third perspective monitors, among many other details, relevant parts for detecting overloading or hardware problems.

The problem statement of this thesis includes the challenging fact that a serverless edge node can easily be overloaded when many triggers occur at the same time. Monitoring the utilization of the node is part of the third monitoring perspective in serverless edge computing. It is a hypothesis, that in case of an overloaded serverless edge node, the serverless functions could be placed at alternative locations for better response times. Monitoring the performance of the nodes themselves is part of the third monitoring perspective of serverless edge computing.

4.3.3 Monitoring Solutions for AWS Lambda

AWS Greengrass is based on AWS Lamda. As a base, this section documents the studies on numerous existing monitoring solutions for the cloud-based AWS Lambda. Section 4.3.4 then studies on the few existing solutions for AWS Greengrass monitoring. In 2016, Yan and Ishakian published, that in their opinion, "*the monitoring infrastructure will be the responsibility of the serverless vendor*" [YCCI16] as he is in charge of the execution containers and is the only party that can monitor every internal aspect of the service. This was also discussed in the section upfront about monitoring perspectives of serverless computing and serverless edge computing.

Today, there are numerous solutions for monitoring AWS Lambda functions that do not fully rely on AWS' internal monitoring. Some of the solutions will be listed below. To summarize their properties, those can be classified into four groups. Firstly, the build-in monitoring solution; secondly, solutions like for instance IOpipe (see Section 4.3.3) that rely on serverless function developers to include their library into the function code; thirdly, tools that use information of the build-in monitoring and provide additional information on top; and fourthly, tools that monitor response times and functionality of the functions without any internal system access by, for instance, sending probe requests to the function in regular intervals. The last group can be seen as the first perspective (end-user perspective) of serverless platform monitoring. The libraries that include code into the serverless functions usually contact external web services on every invocation. They send requests from their own code base to the service without relying on the FaaS provider. Of course, this approach does not guarantee access to all monitorable aspects

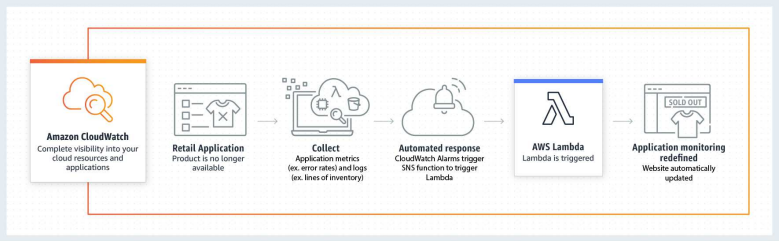


Figure 4.13: Trigger automated CloudWatch Alarms and Lambda workflows to improve customer experience [AWS19b]

of the system that the build-in solution and the solutions on top of build-in provide. For instance, functions that were not able to execute because of overloaded servers, billing or configuration problems would never send requests to an external service.

AWS CloudWatch

AWS CloudWatch is the built-in, cloud-based monitoring and application management service of AWS. AWS CloudWatch can be used for monitoring various of the AWS products like Amazon EC2, Amazon DynamoDB, Amazon S3, Amazon ECS, AWS Lambda, Amazon API Gateway and 70+ more services [AWS19b]. With some modifications, even AWS Greengrass can be used with it. However, it only forwards some local logs and metrics of the edge nodes to the cloud [AWS19b]. "*CloudWatch collects monitoring and operational data in the form of logs, metrics, and events, providing you with a unified view of AWS resources, applications and services that run on AWS, and on-premises servers*" [AWS19b].

AWS CloudWatch can be used to monitor AWS Lambda functions. Table 4.5 lists the metrics that are supported by AWS CloudWatch for AWS Lambda (and was discussed in Section 4.2.2). AWS CloudWatch can be used for analyzing and filtering those metrics. Beside that, any log files that are generated by AWS Lambda can be persisted and also filtered and analyzed in CloudWatch. Figure 4.13 shows that AWS CloudWatch can not only be used for manual queries and analysis but also for automated filters that are able to trigger arbitrary AWS Lambda functions in case of specific log or metric events.

AWS CloudWatch can also be combined with AWS X-Ray. It can be used to trace AWS Lambda functions by giving details of the function invocations and executions [AWS19h]. X-Ray can be bundled with the AWS Lambda code to record outgoing calls, to add additional annotations and metadata [AWS19h]. Additionally, if a Lambda function gets called by another Lambda function, X-Ray traces the requests between them. By the authors best knowledge based on extensive online research, AWS X-Ray is not available for AWS Greengrass in October 2019.

Dashbird

Dashbird is a monitoring framework for AWS Lambda [Das19]. It allows failure detection of the programming code, timeouts and configuration errors. Furthermore, it allows the monitoring of real-time metrics and the system health. It includes monitoring invocation volumes, billed duration, resource usage, errors and alters. Additionally, there is support for analyzing stack traces and X-Ray integration [Das19].

Technically, Dashbird fetches the API data from AWS CloudWatch APIs. Therefore it does not produce any overhead or additional Lambda costs [Das19]. Other monitoring approaches often tend to add additional code to the functions that collects information directly during the invocation and such behavior increases the AWS Lambda execution time and costs. Dashbird on the other hand is only an on-top monitoring service based on the built-in monitoring.

IOpipe

IOpipe provides a framework that wraps Lambda functions and sends runtime metrics to their cloud service. Users of IOpipe can then analyze the requests and metrics of their functions in real time. IOpipe supports various features including statistics, search, web-hooks to slack and other services, error aggregation, tracing of function code and CPU tracing for single code blocks inside a serverless function [Iop18a].

The IOpipe libraries are usually included in the invocation handler of each serverless function. AWS Lambda recently introduced AWS Lambda Layers where function code can be enriched with functionality from other sources - IOpipe was an early adopter of this technology and now provides function monitoring without the requirement of touching function code [Iop18b]. Still, this type of monitoring slightly increases the execution time (and therefore also the costs) of each invocation.

Thundra

The main property of Thundra is monitoring without adding latency or code size to AWS Lambda functions. Other tools like for instance IOpipe (section above) are bundled together with a function and make REST calls during the function invocation [Thu19a]. As mentioned, this adds execution time and uses code space (Mind, that Lambda functions are quite limited in their total code size - see Section 2.2.5 for details).

Thundra provides asynchronous monitoring by collecting CloudWatch logs [Thu19a] - similar to Dashbird. Thundra provides aggregations from the collected logs and supports aggregating and filtering of various metrics including CPU and memory metrics, cache metrics, DynamoDB metrics and domain metrics that can be defined by the developers [Thu19b]. Beside that, the platform includes a service for tracing and debugging AWS Lambda code, a "*warm-up*" service that makes concurrent probe-requests to keep the Lambda containers up and reduces cold starts (see Section 2.2.2 for details on cold and warm starts) [Thu19a].

4.3.4 Monitoring Solutions for AWS Greengrass

The monitoring of Lambda functions on AWS Greengrass devices is, by the best knowledge of the author, not widely explored yet. Built-in support to forward logs to AWS CloudWatch and a local directory, where logs are continuously persisted, exist. To the best knowledge of the author, it is not possible to trigger Lambda functions based on certain metrics or logs. There are no monitoring events that can trigger other Lambda functions based on metrics or logs, which is in contradiction to AWS Lambda (cloud) where this is possible with AWS CloudWatch.

AWS X-Ray is, by the authors best knowledge, also not supported by AWS Greengrass. Monitoring tools, mentioned in the section above, that depend on the inclusion of libraries into the Lambda functions, might work in AWS Greengrass as well. However, these tools - like Thundra or Dashbird - send the monitored data to cloud backends. Sending the data may be impossible in many scenarios of edge computing, as devices can go offline and lack bandwidth at any time. This means that the mentioned monitoring tools for AWS Lambda do not seem reasonable for monitoring serverless functions on the edge.

The built-in logging functionality of AWS Greengrass includes details on the invocation duration of each function as well as the log of each function itself. However, there are no metrics or details about used resources, free resources and the system platform. Section 4.3.5 gives an overview of the different logging components of AWS Greengrass Core.

AWS Greengrass has a build-in forwarding component that forwards the logs to AWS CloudWatch. A caching policy that defines log-rotation and log sizes can be configured. CloudWatch can trigger cloud based Lambda functions if some metrics or log-events occur. Still, those function invocations can only be triggered if the logs were forwarded to the cloud successfully. An edge device may go offline at any time. Greengrass' forwarding mechanism might therefore be unreliable and delayed because of the addressed log-rotation and connection issues.

While there are only limited built-in monitoring features, the online research returned a public Github repository, of Halim Qarroum, that can be used for monitoring some parts of AWS Greengrass - green-sys.

Green-sys

Green-sys is a MIT-licensed open source project by Halim Qarroum [Qar19]. It is a JavaScript based framework for monitoring AWS Greengrass and can be used for "*monitoring of system components (CPU, Memory, Processes, Network, Storage, OS) within AWS Greengrass*" [Qar19], synchronizing system metrics to AWS Cloudwatch, exposing metrics to local and remote MQTT clients and as a remote-terminal-access-service over MQTT that can be used to manage the local Greengrass system from the distance. Qarroums' project is based on *systeminformation*[10], a popular tool for accessing system information and metrics written in NodeJs [Qar19].

[10] https://github.com/sebhildebrandt/systeminformation

4.3.5 Greengrass Structure and Log-File Study

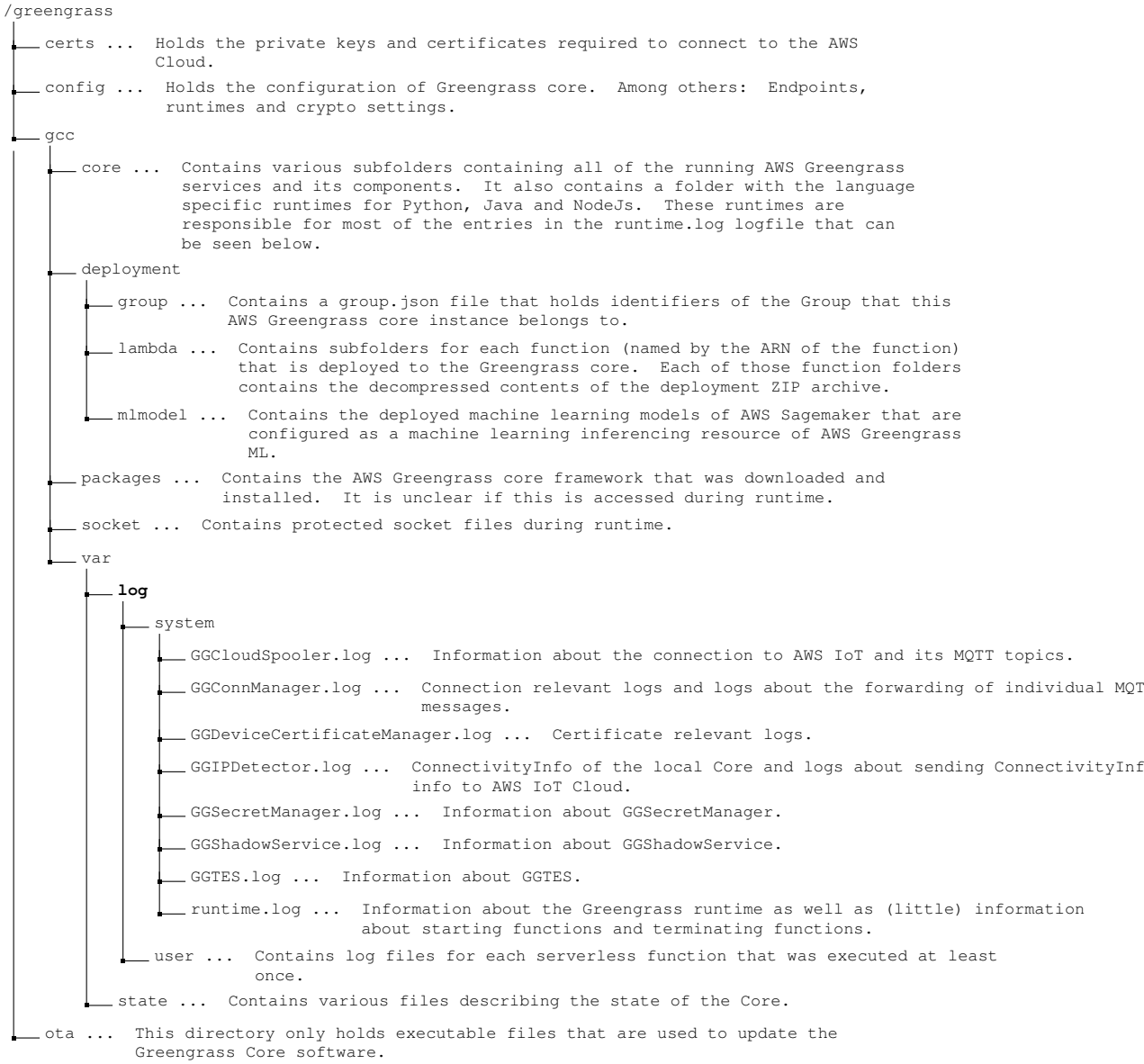
Due to the fact that AWS Greengrass currently, speaking of version 1.9.2, has no sophisticated built-in monitoring tools for the node and function performance, there is no way of monitoring the system utilization with built-in tools. Nevertheless, AWS Greengrass can be configured to keep log-files on the local system and forward them to AWS CloudWatch. As mentioned in the Section above, forwarding may not be reliable due to the edge going offline unpredictably.

This section gives an overview of AWS Greengrass Core's installation directory as well as the relevant log files that are created on the local disks of the edge nodes by AWS Greengrass.

After extensively studying the contents of available log files of a sample instance of AWS Greengrass, it can be said, that the log files contain valuable information about the various components of AWS Greengrass Core itself, including timestamps for starting and terminating function invocations. When an AWS Lambda function writes to its own log, these entries are kept in a separate log directory and file.

By the best knowledge of the author, there are no log-based or trigger-based functions on Greengrass. This means that no other function can be triggered based on logging-events. Metrics regarding the hardware utilization of the Greengrass nodes are not captured with any built-in mechanism - to improve functionality, this needs to be added to AWS Greengrass in future.

Directory Structure The following tree diagram provides an overview of the directory structure of the Greengrass installation directory. The knowledge was gained by manual analysis of installed files. The main focus was on the log files that are created by AWS Greengrass components and on the Python runtimes. It was found that the separate components of AWS Greengrass communicate by a local IPC-socket. Some accessible parts of the inter-process communication was analyzed by sniffing the traffic by using the popular tool *tcpdump*. However, the analysis of the sniffed traffic shows that the gained information is in fact very similar to the information that already gets persisted in the log files of AWS Greengrass by default.



In the tree diagram above, the log files for every component of AWS Greengrass can be seen. Each function and each component also have their own log files. The following screenshot, Figure 4.14, shows a few selected lines of the runtime log of an exemplary function that does nothing but sleep for 10 seconds. The code of the function can be seen in Listing 4.1.

Figure 4.14: Log file for of a Lambda function on Greengrass. Shows an invocation with the generated invocation-ID [125ff1c6-1284-48b7-7fb3-b25a1f3339c0]. Each invocation gets a unique invocation-ID. At line 2, the function gets invoked. At line 14 the function terminates again. The timestamps of each logging entry can be used to calculate the total processing time.

Listing 4.1: rq1-3_helloworldlogging.py

```
1  def function_handler(event, context):
2      client.publish(
3          topic='hello/world/logmessage',
4          payload=json.dumps({'message': 'Sent before sleep!'})
5      )
6
7      time.sleep(10)
8
9      client.publish(
10         topic='hello/world/logmessage',
11         payload=json.dumps({'message': 'Sent after sleep!'})
12     )
13     return
```

Capturing System Utilization As mentioned above, the current system utilization of the AWS Greengrass nodes is not monitored by the build-in tools. There are multiple options for accessing the utilization of a Linux based device. Halim Qarroum's Greengrass-monitoring tool uses the NodeJs based *systeminformation* package for accessing hardware metrics [Qar19]. Various other tools for accessing system metrics also exist. The Python-based PsUtil package as well as accessing */proc/cpuinfo* directly are two examples of this.

Logging the system utilization can happen at three different positions of a serverless edge computing framework:

- Firstly, the metrics could be captured by a service that runs continuously and outside of AWS Greengrass' components. It could capture any system utilization metrics and publish them to AWS Greengrass (as GreenSys does) or put them in a simple log file.

- Secondly, AWS Greengrass could be altered to capture the system utilization metrics as part of one of its components. The metrics could either be persisted continuously, or only when a function gets triggered. In order to make this happen, AWS Greengrass components, like for example ipc-client.py, need to be altered. This is unfortunately not allowed by the AWS License agreements. It is important to mention that it could easily be achieved with AWS Greengrass to persist utilization metrics along the regular invocation logs from Figure 4.14 - this was found by the author during analysis of the files of a freshly installed AWS Greengrass Core node - by altering the runtime code of AWS Greengrass' IPC-Client. Unfortunately, this can not be done without infringing the License §2.c and §2.d of AWS Greengrass: It is not allowed to, nor is it allowed to encourage, assist or authorize any other person to "*modify, alter, tamper with, repair, or otherwise create derivative works of the Software; or reverse engineer, disassemble, or decompile the Software or apply any other process or procedure to derive the source code of any software included in the Software*" [AWS19e]. So this option is not practically or legally feasible.

- Thirdly, the system utilization can also be tracked by each serverless function during its function execution. This can happen by custom logic of the function developer as well as with monitoring tools like Thundra or Dashbird that are included into the function code. Note, that an overloaded node might not be able to start any functions and therefore may lose the ability to monitor the system utilization.

4.3.6 Summary

The three perspectives from which monitoring can be seen in serverless edge computing were discussed. It became clear that - due to the on-premise hosting of serverless edge frameworks - monitoring is the responsibility of the developers themselves.

AWS Greengrass was analyzed for the purpose of finding a proper monitoring interface. Beside log-files there were no findings of other monitoring components.

In conclusion, it was shown that monitoring AWS Greengrass nodes is not straight forward. It seems reasonable to parse the log files of AWS Greengrass in order to get information on the execution time of each serverless function invocation. Additionally, the required information about the system utilization could be gained by altering the IPC-Client of AWS Greengrass and also by sniffing on the network traffic of local AWS Greengrass services. Experiments have shown that this can work, but unfortunately it violates the AWS Greengrass license and is not practical. Storing the utilization metrics as part of every function would work but still requires the functions to start - which would not happen in case of overloading. So, for the sake of stability and license conformity, it would be best to gain the system utilization information with a separate service, outside of AWS Greengrass.

4.4 RQ2: Suitable Machine Learning Approach on Edge

This Section covers the approach to research question 2: "*RQ2: In which ways can a serverless edge computing framework decide where to execute a function (local or cloud) given the current utilization (CPU, RAM, bandwidth, availability of a GPU), QoS (latency, processing time, metrics from RQ1.2)?*" [Section 1.3]

There are several ways how an edge computing framework can decide where to execute a serverless function - for a long period during writing this thesis it was assumed that the serverless function can be executed at the cloud or at the edge.

The decision over where to place the function based on the current system utilization can be a complex optimization problem and could be solved by various approaches. This thesis focuses on finding machine learning procedures that can run on edge devices to re-route requests to the cloud or to process them locally. It was assumed that it could be possible to train a federated learning model on many edge computing nodes that all serve and have to decide over the placement of the same serverless function. But it turned out that the available metrics of AWS Greengrass are not sufficient to train a predictor at all. That is why Research Question 2 can not give an answer on the quality of the proposed ideas and is only used to list some ways to decide where to execute a serverless function given the utilization and QoS of the serverless server.

Section 4.4.1 describes the methodology to tackle the question. Due to the complexity of this tasks, various fields of computer science have influences on the problem. Unfortunately, during the progress of this thesis, it was shown that the idea of using machine learning learning to decide where to execute a serverless function (cloud or edge) is not feasible due to many reasons. The final results of the experiments and literature that justify this statement are given in Section 5.

4.4.1 Methodology

Due to the results of Research Question 1.2, it is assumed that it is feasible to work with AWS Lambda and AWS Greengrass for the further proceeding of this thesis.

In order to find ways of training a predictor locally, another literature and online research was carried out. Due to the size of the whole research area, the focus is limited to existing machine learning frameworks for edge computing and serverless computing. This is, due to the unfortunate reason that many machine learning frameworks do not work together with serverless functions or on the selected hardware platform of this thesis, the Raspberry Pi.

Speaking about hardware, it is an important sub-topic of this research task to find suitable edge computing hardware that can be used for machine learning as well as for serverless computing with AWS Greengrass. Section 4.4.4 lists results of an online recherche about edge computing hardware that is specialized for machine learning tasks. For simplicity (and due to pricing), the chosen platform for this thesis is the Raspberry

Pi 3 because it is cheap and available - further research on this topic should use more specialized hardware of course.

Due to only very few available machine learning frameworks that support edge computing devices for training and are usable within serverless edge computing functions, it needs to be researched which frameworks could even be used for this scope. This was done by an online research and shown in Section 4.4.5. Section 4.4.7 then discusses the suitability of federated learning for edge computing - based on related work, Section 6 finally discusses the suitable and not suitable use cases of edge analytics and federated learning in edge computing.

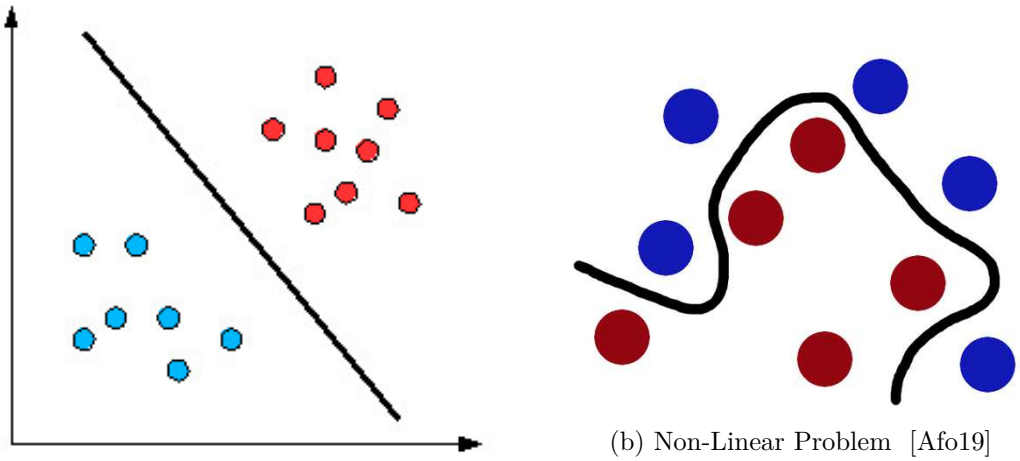
Section 4.4.2 is based on relevant literature and gives details on machine learning basics that are relevant for building a predictor. Section 4.4.3 describes training data that could be used for training the predictor - it is based on findings of research question 1.2 and 1.3 and also on some experiments done in Section 5. Section 4.4.4 provides the results of the online analysis on edge computing hardware with built-in support for machine learning. Section 4.4.5 gives details on doing machine learning on edge devices and Section 4.4.6 gives details about doing so within serverless functions. Section 4.4.7 discusses the promise of federated learning and how it could help in edge computing.

4.4.2 Machine Learning Basics

Regression vs. Classification According to Marsland, "*we need to separate out regression problems, where we fit a line to data, from classification problems, where we find a line that separates out the classes, so that they can be distinguished*" [Mar09]. But due to the following, this does not matter much: The conducted literature analysis on machine learning brought up that classification problems can as well be seen as regression problems and vice-versa. "*However, it is common to turn classification problems into regression problems. This can be done in two ways, first by introducing an indicator variable, which simply says which class each datapoint belongs to. The problem is now to use the data to predict the indicator variable, which is a regression problem. The second approach is to do repeated regression, once for each class, with the indicator value being 1*" [Mar09].

Linearity vs Non-Linearity The conducted literature analysis about relevant machine learning facts brought up that Classification problems can be divided into linear and non-linear problems and that linear separable problems will work much faster on devices with low resources (what edge devices typically are):

The training process of a classification algorithm usually results in a predictor that separates the data points by a line or more general by a hyperplane. Cases where data can be separated by a straight line are called linearly separable case, while others are called non linearly separable. Figure 4.15a shows linearly separable data and Figure 4.15b shows linearly inseparable data [Afo19]. It is clear to see that some data sets can be described with linear functions while others require more complex ones.



(a) Linear Problem [Afo19]

(b) Non-Linear Problem [Afo19]

Figure 4.15: Linear and Non-Linear data sets [Afo19]

Linear separable models have various advantages over non-linear models [JWHT14]. James et al. name the fewer needed computational resources for inferencing and the simplicity in describing and interpreting the models as main advantages of of linear models over other approaches [JWHT14].

4.4.3 Training Data

What to measure? Research Question 1.2 listed suitable metrics for monitoring the performance and Quality of Service (QoS) of a serverless edge computing function. The metrics found to be most important in edge computing are Latency, Bandwidth and Computational Load (especially CPU, RAM and network utilization but based on the individual task, other metrics like disk IO, GPU or other device metrics might have higher entropy).

The latency of a request usually depends on two factors, the network delay between devices and the computational delay that comes from completing a computational task. The network delay can only reasonably be measured on a client device. A device that runs a serverless edge computing framework can itself not measure the network latency intrinsically. For having measurements about the network latency at the serverless edge, it would need to get those measurements sent by a client device. But an IoT client is usually very simple, energy-efficient and therefore not suitable for doing that.

If system performance gets worse, computational latency increases and the overall latency (computational plus network latency) increases too. This was stated by Baresi et al. [BMG17] and also confirmed by the experiments of this thesis (see Section 5).

For a first prototype it seems reasonable to train a computational-latency-predictor that uses CPU and RAM utilization of a serverless edge node as training data - a supervised

learning problem (for machine learning background see Section 2.5.2). This use case is also very handy because the labeled training data that is required for this process could typically be generated by the edge computing framework - by just measuring its invocation duration. This is reasonable because, traditionally, those two resources impact the system performance most and are very simple to measure on Linux based systems. In state of the art applications, other hardware metrics might also be valuable. For example, GPU utilization, disk-io and others can be considered in another research paper with a wider scope. When a serverless function heavily depends on disk usage, network bandwidth, GPU resources or others, results might differ and need to be checked separately.

A data set usually consists of many training samples that have multiple features each. These features can be continuous, categorical or binary [MKWS07]. If every instance in the data set "*is given with known labels (the corresponding correct outputs) then learning is called supervised, in contrast to unsupervised learning where instances are unlabeled*" [MKWS07].

In Section 5, a prototype was implemented to simulate a scenario where an edge device is overloaded by too many requests. Overloading was expected to be measurable by CPU and RAM utilization as well as the response time (network latency) of client devices. Furthermore, with another prototype it was investigated if forwarding invocations of AWS Greengrass functions to the cloud-based AWS Lambda is reasonable. For the whole process of this thesis it was expected that, when forwarding, the computational latency would decrease and the network latency would increase in a way that the total network latency, that is measured at client devices, decreases. But the findings actually indicate that forwarding requests with AWS Greengrass is extremely inefficient and furthermore, it was found that high CPU usage on Greengrass does not indicate that AWS Lambda would always return a result in less time. Details can be found in Section 5.5.

In order to train an actual predictor, data of CPU and RAM utilization for a given serverless function invocation need to be available as well as data about the execution time of a Lambda function need to be there. The next paragraph gives details on why it was not possible to get such training data and why the required information can not be gotten out of AWS Greengrass.

How to acquire? The analysis for Research Question 1.3 shows that acquiring CPU and RAM metrics for AWS Greengrass is not straight forward. Acquiring the data is difficult because in order to find the average execution time of a serverless function based on a given system utilization, these data points need to be persisted in some manner first. Unfortunately, AWS Greengrass does by default not support persisting any utilization metrics or execution times. This is in total contrast to AWS Lambda where RAM usage and execution duration is persisted into log files by default.

Unfortunately, there are not many options for acquiring the utilization metrics on Greengrass. Section 4.3.5 lists the options. Summarizing the options: Adding additional

code to every serverless function. Altering Greengrass' core code. Having a service that is running beside Greengrass.

None of those options is ideal! Altering Greengrass' core code is permitted by the license of AWS. Having a second service on the edge that continuously logs the utilization produces some computational overhead and matching AWS Greengrass built-in log files with the service can be error-prone. And using the serverless functions themselves for logging the utilization - with e.g. wrappers - adds additional overhead to the processing time of each function run.

So it turned out that acquiring performance metrics on AWS Greengrass is not straight forward. Furthermore, during implementation of some prototypes (see Section 5) it turned out that AWS Greengrass also logs wrong values about the duration of each function invocation. That means that not only the performance metrics can hardly be measured, also the QoS of each function invocation can not be measured on AWS Greengrass. Details about that can be found in Section 5.3.

4.4.4 Suitable Edge Machine Learning Hardware

Due to the limited computational resources of edge computing devices, machine learning and data analysis tools from cloud computing might not properly work on resource constrained edge devices. But recently, some of the mentioned frameworks started to support special hardware that accelerates their performance on edge devices compared to a traditional CPU.

While, in theory, *the serverless edge* can be seen as various different hardware setups (Ishakian et al. perform their deep learning experiments in three environments, locally on device, local data center and cloud [IMS17]), larger vendors like Google, AWS and Microsoft use embedded devices like Raspberry Pi or Nvidia Jetson as their default hardware. This section gives an overview over the most recent advances in edge computing hardware.

Back in 2005 it was shown that some machine learning algorithms and especially neural networks run significantly faster (back then up to 3 times) on GPUs than on CPUs [SBS05]. Current machine learning frameworks support and prefer processing on GPUs.

To use this machine learning and existing frameworks on edge devices there will naturally be a demand for either CPU based solutions or increasing GPU power on the edge. Current edge devices (like AWS Greengrass' default device; The Raspberry Pi 3) do not have dedicated GPU processing power that is usable with current frameworks. Recently, some mobile phones are produced with additional built-in processors that support "*on-device*" machine learning workflows. Apple first used those chips in their iPhone X, 8 and 8 Plus series in September, 2017, within their A11 Bionic chip [App17]. Various hardware manufacturers are also currently launching GPU units that are suitable for edge computing. The following sections will shortly introduce and compare some of these hardware enhancements.

ARMs Project Trillium In February 2018, ARM announced its Project Trillium (brand name will be changed in the future). A set of new processors that will deliver machine learning and neuronal network functionality specifically designed for inference at the edge. Their goal is to maintain thermal and power efficiency while exceeding the performance of standalone CPUs, GPUs and accelerators. "*While the initial launch focuses on mobile processors, future Arm ML products will deliver the ability to move up or down the performance curve – from sensors and smart speakers, to mobile, home entertainment, and beyond*" [ARM18a]. ARM announces a performance of "*up to 4.1 TOPs, with a stunning efficiency of 3 TOPs/W for mobile devices and smart IP cameras*" [ARM18b].

Rockchip RK3399Pro Rockchip is works on a similar platform to ARMs Project Trillium. It is reported that this platform's performance reaches up to 2.4 TOPs by a power consumption of "*less than 1%, comparing with other solutions adopting GPU as AI computing unit.*" [New18]. Furthermore, it supports existing machine learning frameworks for inferencing such as TensorFlow Lite [New18].

Google Edge TPU Google is currently (September 2018) running an early access program of Cloud IoT Edge Alpha and Edge TPU Early Access development boards, their edge computing based machine learning inferencing accelerator [Goo18c]. The Edge Tensor Processing Unit (TPU) "*enables users to concurrently execute multiple state-of-the-art AI models per frame, on a high-resolution video, at 30 frames per second, in a power-efficient manner*" [Goo18c].

Accordingly to Google, there is an increasing demand for cloud trained AI models to be run on the edge. They announced that "*Edge TPU is Google's purpose-built ASIC designed to run AI at the edge. It delivers high performance in a small physical and power footprint, enabling the deployment of high-accuracy AI at the edge*" [Goo18c] .

Nvidia Jetson The Jetson product line of Nvidia provides low power devices for machine learning applications. While the first device of the series (Jetson TK1) was published in 2014, there are already two successors: TX1 and TX2. In August 2018, Nvidia announced Jetson Xavier as a new successor[11].

The Nvidia Jetson TX2 is a 7.5 watt, credit card size computer that is equipped with an Nvidia Pascal-family GPU and 8GB of working memory, Gigabit Ethernet, 802.11ac Wifi and Bluetooth[12] .

It is said to be able to detect cars, bikes and people from 2 simultaneous video inputs with a HD video resolution each[13]. Furthermore, there is official support for AWS Greengrass to be used for inferencing AWS Sagemaker models [AWS19g].

[11] https://developer.nvidia.com/embedded/buy/jetson-xavier-devkit
[12] https://directory.ifsecglobal.com/40/product/95/41/68/327494_DS_Jetson_TX2_Module_A4_fnl_WEB.pdf
[13] https://www.zdnet.com/article/nvidia-takes-pascal-onto-jetson-to-boost-embedded-ai/

Intel Movidius Intel did recently join the market of specialized machine learning processors with its Intel Movidius. The Intel Movidius Neural Neural Compute Stick is an USB device with an integrated processor for accelerating machine learning on edge devices. It features the Intel Myriad 2 processor that, according to Intel, is capable of up to 4 TPUs and has a low energy use [Int18].

Raspberry Pi 3 The Raspberry Pi does not support any hardware accelerators for machine learning. Still, due to its quad-core-processor and one gigabyte of working memory it might be possible to use the device for machine learning inference. AWS lists the Raspberry Pi 3 as their default hardware platform for serverless edge computing with AWS Greengrass.

Others There are many other hardware platforms that use similar (sometimes almost identical) software and hardware to the Raspberry Pi. Worth mentioning are Orange Pi, ODROID-C2, BANANA Pi, Beagle Board, Intel Edison and Arduino. They differ in specific hardware details like working memory, energy consumption, port interfaces and processor power. In favor of the previously mentioned advanced edge devices, such other devices are not taken into consideration in the scope of this thesis.

4.4.5 ML-Training on Edge Devices

During the research progress of this thesis, it has turned out that *using machine learning* on edge devices needs to be separated into two different categories: Machine Learning Training and Machine Learning Inferencing. Training describes the process of creating a machine learning model based on training data. Inferencing describes the process of using a previously created model to process ("infer") a new data sample in order to classify the new input.

As already elaborated in Section 2, it is state of the art to use edge devices for machine learning inference. Section 3 listed many projects that already work with inferencing on the edge devices. The underlying models are often not trained on edge devices, instead the models are usually trained at cloud data centers. The next paragraphs summarizes some exemplary projects that use machine learning inferencing on the edge already.

In general, running existing machine learning frameworks on Edge devices is not straight forward because existing frameworks are often either not compatible with edge computing hardware and there are also often incompatibilities based on the available operating system libraries of edge devices e.g. AWS Greengrass did not support Python 3 until Summer 2019.

Greengrass ML As mentioned in various Sections above, AWS Greengrass is a serverless edge computing framework that is based on AWS Lambda. Section 4.1.3 gives an overview over AWS Greengrass. Greengrass ML is an extension of AWS Greengrass that allows serverless AWS Lambda functions that are deployed on AWS Greengrass to

access machine learning models that are managed with AWS Greengrass ML. "*AWS IoT Greengrass makes it easy to perform machine learning inference locally on devices, using models that are created, trained, and optimized in the cloud. IoT Greengrass gives you the flexibility to use machine learning models in Amazon SageMaker or to bring your own pre-trained model stored in Amazon S3*" [AWS19g].

AWS Greengrass officially supports NVIDIA Jetson TX2 as an edge device, meaning that inferencing can be accelerated with the Machine Learning accelerators of NVIDIA.

Unfortunately, Greengrass ML only supports the deployment of trained models and does not training models from scratch or training online-learners that incrementally adapt to new data. Some experiments of the author have shown that AWS Lambda and AWS Greengrass functions, support the popular machine learning framework Scikit-learn which could then be used for basic model training. But it needs to be mentioned that Scikit-learn is only used for very basic machine learning operations and does not even support running on a GPU [SL17b].

TensorFlow Light The regular edition of TensorFlow can be used for Machine Learning Inferencing as well as for Machine Learning Training. But, TensorFlow was not a good choice for edge devices because it had no official support for ARM processors until recently. As discussed in Section 4.4.4, there are not many edge devices that are not based on ARM- CPUs. The Raspberry Pi is the default edge device of most common serverless edge computing platforms (AWS Greengrass and Microsoft IoT Edge) [AWS18c] [Mic17b].

One can argue that TensorFlow supports ARM processors well with its fork called TensorFlow Light [Lit19b]. "*TensorFlow Lite is a set of tools to help developers run TensorFlow models on mobile, embedded, and IoT devices. It enables on-device machine learning inference with low latency and a small binary size*" [Lit19a].

TensorFlow Light is used in popular products like G-Mail, Google Photos, Nest, Google Assistant, and VSCO [Lit19b]. But TensorFlow Light is not equal to the regular TensorFlow. It states to be an inferencing framework. Technically, a already trained TensorFlow model (trained with the regular TensorFlow) will be converted into a compressed flat buffer with the TensorFlow Lite Converter [Lit19b]. This flat buffer can then be deployed to Mobile Phones and IoT devices including Raspberry Pi, iPhone and Android Phones [Lit19b].

"*TensorFlow Lite does not currently support on-device training, but it is in [their] Roadmap, along with other planned improvements*" [Lit19a].

Federated Learning is referenced in some Related work in Section 3 in this thesis. There is a sub-project of TensorFlow that supports Federated Learning. This project is called TensorFlow Federated. Unfortunately, this framework does not yet support Raspberry Pis [Ost19] and "*only supports simple local simulations*" [Ost19] and can therefore not be used on edge devices (yet).

Flogo The serverless computing framework Flogo was already discussed in Section 4.1.3. There, an interesting feature was mentioned: Flogo supports Machine Learning Inferencing with the help of Tensorflow Light [Flo19a]. Unfortunately, they only support inferencing and not model training. Details about Flogo Inferencing can be found on the website of Tibco [14] who is also the main sponsor of Flogo. There they state that "*The inference activity was built to support the concept of plugable frameworks, however the only supported framework is currently TensorFlow. The activity leverages the Golang API from TensorFlow. You don't need Python or anything other than the TensorFlow dynamic library installed on your dev & target machine* [Flo19c].

Azure Machine Learning on Edge Microsoft Azure IoT Edge support deployments of serverless functions as well as deployments of Microsoft Azure ML machine learning models [Mic17d]. Similar to the products above, Azure IoT Edge ML does only support inferencing of pre-trained models - no training [Mic19].

Scikit-learn Scikit-learn is a basic machine learning framework for simple machine learning workloads. It does only support a limited amount of machine learning models but at least it supports online-training with its *partial-fit* method for some models like Stochastic Gradient Descent and Multilayer Perceptron. The author of this thesis tested Scikit-learn on AWS Lambda and AWS Greengrass. While it just works, there are many disclaimers at the project documentation that state that Scikit-learn is only made for educational purposes and not ready for production use. Scikit-learn does not even have support for GPUs to run machine learning code faster - nearly any production ready machine learning framework like TensorFlow, Apache MxNet, PyTorch have support for GPUs. But regarding GPU use, Scikit-learn's official statement regarding GPU support is: "*No [support], or at least not in the near future. The main reason is that GPU support will introduce many software dependencies and introduce platform specific issues. scikit-learn is designed to be easy to install on a wide variety of platforms. Outside of neural networks, GPUs don't play a large role in machine learning today, and much larger gains in speed can often be achieved by a careful choice of algorithms*" [SL17b].

Android Neural Networks API "*The Android Neural Networks API (NNAPI) is an Android C API designed for running computationally intensive operations for machine learning on Android devices. NNAPI is designed to provide a base layer of functionality for higher-level machine learning frameworks, such as TensorFlow Lite and Caffe2*" [Goo19]. NNAPI supports inferencing by applying data from Android devices to previously trained machine learning models, NNAPI does not support training models, it only supports using the weights of previously trained models. "*NNAPI is meant to be called by machine learning libraries, frameworks, and tools that let developers train their models off-device and deploy them on Android devices. Apps typically would not use NNAPI directly, but would instead use higher-level machine learning frameworks.*

[14] https://tibcosoftware.github.io/flogo/development/flows/tensorflow/inferencing-tf/

These frameworks in turn could use NNAPI to perform hardware-accelerated inference operations on supported devices" [Goo19].

4.4.6 ML within Serverless Functions

As outlined in Section 4.4.5, running machine learning frameworks on edge devices is difficult because of resource limitations. This section outlines running machine learning in serverless functions which, as it turned out during multiple code experiments as part of this thesis, is even more difficult because of, different, resource limitations.

Machine Learning Training usually requires longer running time. Serverless functions are usually limited in their maximum execution time. AWS Lambda is for example limited to 900 Seconds of maximum function runtime [AWS17]. 900 Seconds is just by far not enough to train some popular machine learning models in batch training mode. The serverless edge computing framework AWS Greengrass is special in this case because it supports so called long-running-serverless-functions that are not limited in execution time [AWS18c]. They start once and then stay running until the code of the function finishes or AWS Greengrass shuts down.

So in theory, AWS Greengrass would be more suitable for machine learning training than AWS Lambda if only maximum execution time would matter. But AWS Greengrass is limited in other ways. Until summer 2019, there was no support for Python 3 on AWS Greengrass. Many machine learning frameworks have recently stopped supporting Python 2. Furthermore, AWS Greengrass' default edge device is the Raspberry Pi, an ARM-based microcomputer. Unfortunately, many machine learning frameworks can not be compiled for this architecture yet. This means that the execution hardware limits the serverless function code that is running on it. Cloud-based AWS Lambda functions can - in theory - be used equally on AWS Greengrass. But, as found by the author of this thesis, as soon as the serverless function packages some dependencies (e.g. im Python a library like Numpy) in its deployment package, each packaged dependency must fit to the hardware of the device that is running the serverless function later. During implementation of this thesis it turned out that this would limit the possibility of dynamic redeployment of serverless functions between Raspberry Pi based Greengrass nodes and x64 based AWS Lambda cloud servers. One would need one version of a function for AWS Lambda (with x64 based dependencies) and multiple versions for AWS Greengrass depending on the architecture Greengrass is installed on (can be x64 or ARM).

Beside the constraints that come from hardware differences, AWS Lambda functions are limiting machine learning in another way: During implementation of this thesis it was a problem that AWS Lambda limits the size of the deployment packages for a serverless function with 50MB. This means that the serverless function with all its dependencies and required libraries needs to fit in 50MB (zipped. 250MB unzipped). Many python-based machine learning frameworks depend on, for instance, Numpy and Scipy which together nearly hit this 50MB-limit (zipped). Additionally, packaging scikit-learn with such a function makes it to big for an AWS Lambda function.

There are various ways to circumvent the code size limitation on AWS Greengrass. By installing required machine learning dependencies on the Raspberry Pi itself makes those dependencies available for every AWS Lambda function that runs on this device - the local Python runtime then just has the required dependencies available. This was tried and done by the author while implementing some of the experiments, but this circumvent stands in clear contradiction to the serverless computing paradigm of having a capsuled serverless function, that brings with its requirements and that does not depend on the executing server it's running on. A circumvent like this will obviously not be possible on the cloud-based AWS Lambda platform because developers can not access the servers the functions are running on. So this also results in serverless functions that can only be used at one specific device type. There is currently no support to package serverless functions for different hardware platforms and with additional dependencies. Having support for both of this is probably a necessity for bigger deployments. Docker, for instance, has support for different hardware-deployment-packages for the same artifact name[15].

Another problem of machine learning within serverless functions is lacking support for GPU accelerators. The market overview of the biggest cloud platforms for serverless computing in Section 2.2.5 has shown that there is no platform that supports GPUs in any of their serverless computing framework yet. As exceptions confirm the rule, AWS Greengrass is one exceptions and officially supports machine learning accelerators for machine learning inferencing at edge devices from serverless Lambda functions. But this interface is limited to inferencing only. To have better support for machine learning in serverless functions, regardless if edge based or cloud based, those functions need to be able to access GPUs or FPGAs natively or in a shared manner.

The already mentioned limited maximum execution time of a serverless function does also limit model training of bigger models. Therefore, batch training is a exceptionally bad fit for serverless computing. Only those concepts of training machine learning models incrementally or in parallel make sense to be used with serverless computing. A literature analysis was done to identify those concepts and the results include out-of-core learning [Gro17], online learning [Gro17] and Federated Learning [MMRyA16].

4.4.7 Federated Learning in Edge Computing

This thesis summarized various related work from the fields of edge computing, serverless computing and machine learning in Section 3. The related research has shown that Federated Learning is a promising approach for distributed machine learning in an unstable environment - as for example edge computing [MMRyA16, WHW+18, SL17a]. This is why it was assumed that Federated Learning can also benefit the problem of predicting serverless function latency on edge devices.

To summarize, Section 4.1.2 elaborates that there are currently no serverless edge computing frameworks with built-in support for Federated Learning. Still, the idea of

[15] https://engineering.docker.com/2019/04/multi-arch-images/

Federated Learning could be a great fit for edge [WHW+18]. It can be seen as an advancement of incremental online learning.

Federated learning proposes to solve the problem of having separate models on various devices by introducing a parameter server. The parameter server is the center of the distributed network and occasionally receives the updates of individually trained machine learning models from the participating devices. The server then combines those models into a collective model and distributes the merged model back to the individual edges. There, the models will be further trained based on the available training data on those edges. The parameter server never receives any training data samples from the participating individual edges - only the weights and parameters of the machine learning models. For Details on how Federated Learning works see Section 3.3.

The main advantage of Federated Learning over traditional machine learning approaches is that the machine learning model is trained as sub-models on separate machines with separate non-idd data. The resulting sub-models will then be sent to a central so called parameter server and be merged with a federated learning strategy like averaging [MMRyA16]. The averaged models are then distributed back to the individual edges and further trained with additional data points. The crux of the matter is that the training data is never sent to the cloud - meaning that the required bandwidth is much lower. Furthermore, federated learning has various other benefits including enhanced privacy in comparison to traditional centralized training approaches because the training data stays at the edge devices and never goes to public cloud data centers. Bonawitz et al. [BIK+17] have shown that federated learning can be enhanced to better protect the privacy of each learning-node and they request further research on this topic.

Section 3.3 summarizes the findings of various publications of distributed machine learning and federated learning. Some of them also elaborate on edge computing explicitly. Their main findings are, beside privacy enhancements, that the bandwidth that is used to transfer all data from client devices to a cloud-based ML-training service is much higher than the required bandwidth if the data is already used for training at edge devices [MMRyA16]. Speaking about bandwidth, its much more efficient to only send the updated machine learning models from and to the cloud-based parameter server.

Of course, this reduction in bandwidth has more impact if the file size of the training data samples is bigger. For the experiments of this thesis, Federated Learning might not be a good fit because each training data sample is very small because it only involves numbers representing the execution duration as the label for the supervised machine learning problem as well as the CPU utilization and the RAM utilization.

Cloud data centers have super short network distances between server instances and are equipped with powerful hardware, edge computing nodes are typically located close to users and therefore have longer network distances between other servers and are limited in energy consumption and hardware. While the network connections between servers of cloud data centers is usually not much limited in bandwidth, the network connection of edge computing nodes can be very limited. For instance, take an edge computing node

that is located at a factory building in rural areas of Austria - the internet connection is often not very powerful there. Federated Learning could also help in such scenarios. But in order to do that, significant computational power is needed at the edge. Its always an offtrade between network bandwidth and computational power. This is further going to be discussed in Chapter 6.

4.5 RQ3: How well is the processing time predicted?

This section gives an overview over the analysis of Research Question 3: "*To what extent does edge analytics improve the latency and overall processing time of serverless (edge) functions? In which ways does the suggested approach (for selecting the destination of a function) change the performance of the function. Discussion of the results is required.*" [Section 1.3].

4.5.1 Methodology

Research Question 3 will not be analyzed by any literature analysis. Rather, it will be shown, how well edge analytics can be used and in which ways *selecting the destination of a serverless function* can be done, by the outcomes of implementing some prototypes and various experiment. The results and exact procedure of this are listed in Section 5.5.

CHAPTER 5

Implementation and Results

This chapter lists all results of this thesis and discusses them. Chapter 6 is used to discuss the implications of these findings. Chapter 7 summarizes the whole thesis.

In order to structure this chapter, the following subsections discuss the findings based on the initially defined research questions from the proposal of this master thesis.

5.1 RQ 1.1 - Suitable Frameworks Results

"*RQ1.1: What are suitable frameworks for serverless edge analytics? Which parts of the suitable frameworks need to be used and which modifications are required to support machine learning on these platforms?*"

Research Question 1.1 was answered by obtaining details about a number of available serverless frameworks, edge computing and requirements for machine learning in Section 4.1 and by gaining insights during implementation of the prototype code. Consequentially, some criteria for defining *suitable* frameworks were defined. Those are listed in Table 4.1 and for easier reading, Table 5.1 shows the same information again. Details about the Table and the requirements can be found in Section 4.1.

To summarize those requirements: Not all frameworks are able to run in edge environments due to restrictions regarding their environment - that is the Edge Requirement. Edge Computing is often used in IoT environments and therefore at least some of the identified interaction paradigms of the IoT, like MQTT, should be supported. The Community Requirement regards the data science community whose demands were found by survey with thousands of participants done at Kaggle [Kag17]. This study has shown that some frameworks and programming languages are crucial for data scientists and should therefore be supported by a serverless edge computing framework in order to attract this community. The Inferencing Requirement lists if the framework has built-in capability to use pre-trained machine learning models for inferencing. Even in cloud-based setups,

machine learning is usually executed in MLaaS environments and only accessed from the serverless functions and not within serverless functions. Some frameworks support a stripped-down version of MLaaS on the edge. As this is an enabler for many analytic use cases on edge devices, this is called the Inferencing Requirement. The Training Requirement, on the other hand, states the explicit support to train machine learning models on the serverless edge computing node. This includes various smaller requirements like available frameworks, hardware, the capability to provide data for such an algorithm and monitor results. Clearly, as of today, non of the frameworks supports this out of the box. For AWS Greengrass it was shown that Scikit-learn can be used within its serverless functions but this is very limited for practical use. The Maintainer Requirement describes that a framework should be actively maintained because, as it turned out during this thesis, there exist various tools that are abandoned and not taken care of by its developers anymore. The Hardware Requirement was identified due to the demand of GPUs and machine learning accelerators for many machine learning frameworks. An overview over available edge computing hardware was given and unfortunately, so far, only AWS Greengrass supports accessing those accelerators and, even worse, only for inferencing. The Execution Requirement comes from the demand of machine learning training for long running, resource intensive code with complex code dependencies and rather big deployment package size. This means that the framework needs to support an execution environment where those machine learning processes can actually be executed. The Synchronization Requirement describes the need of synchronizing machine learning models between different edge computing nodes. Non of the frameworks brings an actual built-in solution for that. Only AWS Greengrass and Microsoft IoT Edge support some rudimentary, very constraint synchronization of states. There is actually no suitable built-in solution available to actually synchronize edge-trained machine learning models or do Federated Learning without mayor adaptations.

	FLO	OW	OWL	MIE	AGG
Edge Requirement	Yes	No	Yes	Yes	Yes
IoT Requirement	Yes	Yes	Yes	Yes	Yes
Community Requirement	No	Yes	Yes	No	Yes
Inference Requirement	Yes	No	No	Yes	Yes
Training Requirement	No	No	No	No	Partly
Maintainer Requirement	Yes	Yes	No	Yes	Yes
Hardware Requirement	Partly	No	No	No	Yes
Execution Requirement	No	Yes	Yes	No	Yes
Synchronization Requirement	No	No	No	Partly	Partly

Table 5.1: Duplication of Table 4.1 for easier reading. Summary of the fulfilled requirements for each framework. Flogo (FLO), OW (Apache OpenWhisk), OWL (Apache OpenWhisk Light), MIE (Microsoft Azue IoT Edge), AGG (AWS Greengrass)

The outcome of this research, based on the requirements, is that AWS Greengrass can

be considered as the most mature, mayor serverless edge computing platform today and that it is, therefore, suitable for further experimentation. Based on the found criteria, Greengrass has various advantages over its rivals - those include a very efficient implementation, maintenance, good hardware support and capabilities for machine learning inferencing. In regard of suitability for machine learning training, non of the frameworks is actually regarded suitable. Regarding machine learning inferencing, Flogo and Microsoft Azure Edge need to be named.

Beside this main finding it was also found that all serverless edge computing frameworks support various interaction paradigms of the Internet of Things. Furthermore, it was found that there is no software-support for ML-training at any available serverless platform. Beside that, available edge computing hardware, see Section 4.4.4, only rarely supports accelerated machine learning for inferencing and even worse, non of them was actually identified to support machine learning training.

This brings us to the discussion about the second part of research question 1.1, regarding support of machine learning and federated learning on serverless edge platforms and which modification would be required. Some of the evaluation criteria mentioned above were chosen especially for evaluating machine learning capabilities and to identify missing features. The outcome of the literature and online analysis done for this master thesis is, that with current technology, edge devices can be impressively used for using ML-inferencing but are not yet capable enough for ML-training. There is the problem of monitoring machine learning training at the edge and there is no suitable hardware supporting machine learning training. Furthermore, advanced distributed machine learning techniques like hypothesis transfer learning and federated learning (see Section 3.3) are yet not fully available for edge devices and also not for serverless devices.

Another finding is that there is no (cloud based) serverless platform that supports machine learning accelerators inside of serverless functions today. Some cloud based serverless frameworks support external machine learning tools that can be invoked and accessed from within serverless functions but those tools do not run machine learning code inside a serverless runtime - e.g. AWS Sagemaker can be invoked from AWS Lambda.

The identified mayor drawbacks of running machine learning in serverless functions mainly are: missing hardware support and constraints of serverless functions regarding maximum code size and maximum execution time of serverless functions. Details about this can be read in Section 4.1. Furthermore, during literature analysis about Federated Learning it turned out that the Federated Learning implementation of TensorFlow is not able to run on ARM based processor architectures yet [Ost19]. This is, regardless if serverless or not, problematic in the context of edge computing because, as identified in the online research about edge hardware in Section 4.4.4, many edge devices are actually ARM-based.

5.2 RQ 1.2 - Metric Results

RQ1.2 What are means and suitable metrics for monitoring the performance and Quality of Service (QoS) of a serverless edge computing function? How does this differ from ordinary serverless functions?

In order to answer Research Question 1.2, further literature analysis was done. The analysis can be found in Section 4.2. The scope of the analysis was finding metrics that are common in at least one field of cloud computing, edge computing and serverless functions. Furthermore, a discussion about the difference of suitable metrics for serverless functions on the edge versus serverless functions in the cloud was given in Section 4.2.3.

The suitable metrics for monitoring serverless edge computing functions are discussed in Section 4.2.3 and Section 4.2.4. Latency, especially End-to-End-Latency, was identified as one of the most important metrics among bandwidth and computational load. Furthermore, other metrics, like, for example, metrics regarding the invocation payload and metrics for analytical functions like GPU utilization, were identified to be valuable for some use cases like machine learning and dynamic rescheduling of functions. More about that can be found in Section 4.2.4.

Another result is, that serverless edge computing requires more performance metrics than cloud computing because in contrast to cloud based FaaS services, serverless edge computing was found to be only available on-premise/selfhosted yet. This means that performance needs to be measured for the individual function invocations (same as for cloud based serverless computing, the FaaS perspective) and additionally, the edge devices that host the serverless frameworks need to be measured precisely because they can not scale as easily as cloud data centers (IaaS perspective). Beside that, it was found that for AWS Greengrass, the total request latency can only be measured effectively by end devices, computational load can be measured directly on the edge devices but not with AWS Greengrass itself - more about that is listed in Section 5.3.

The carried out literature analysis undermines the requirement for metrics about the health of the serverless edge computing hosts. Baresi et al. found, that Edge Computing can drastically increase system performance as long as the edge device is not overloaded [BMG17]. As soon as overloading happens, processing time and latency increase into unacceptable sizes [BMG17]. That means, that metrics that are collected for serverless edge computing functions can not only be focused on the function invocations and request latency of each serverless function invocation. This is the case for FaaS based serverless computing but not for edge-based. Metric collection on serverless edge devices needs to be seen wider than on cloud-based FaaS services because there is no FaaS on the edge. As mentioned, every framework is self-hosted. If an edge device starts to be overloaded, requests should be sent somewhere else - otherwise, later on, the performance metrics for the serverless functions will reveal worse performance - but they can only be sent somewhere else if its detectable that the nodes health is getting worse.

The background research about current state of the art in Chapter 2 has brought up that all serverless edge computing solutions are currently self-hosted, on-premise. Due to that fact, edge based serverless deployments need to keep track of metrics that are important for monitoring serverless edge computing nodes including, infrastructure metrics, hardware utilization metrics of the edge (similar to IaaS metrics) as well as the execution metrics for each serverless function invocation (like startup time, coldstart/warmstart, payload size, latency), current serverless edge computing frameworks can be seen as its own type of cloud computing as it is a mix between FaaS, SaaS and IaaS.

To summarize the findings, it can be said that for ordinary (cloud-based) serverless functions, only metrics regarding the used resources, performance, costs and execution time of the function matter. For serverless edge computing, on the other hand, the same metrics as for cloud-based functions are relevant but furthermore, metrics regarding the actual hardware that runs the serverless framework are important. That is because serverless edge computing frameworks are until today not provided as FaaS.

5.3 RQ 1.3 - Monitoring Results

RQ1.3: How can serverless edge computing devices and the corresponding function runs be monitored and stored (e.g. in AWS Greengrass)? Which tools are required?

5.3.1 Findings from analysis

To answer Research Question 1.3, various tools and monitoring frameworks were identified from literature in Section 4.3.3. Because AWS Greengrass was found to be the most suitable framework by Research Question 1.1, the monitoring capabilities of AWS Greengrass were looked at in detail. It turned out that monitoring AWS Lambda function runs on AWS Greengrass is not well supported yet. Details about this and about monitoring approaches in general will be given throughout this section.

Section 4.3.2 was used to describe the analysis on how to monitor serverless functions. Firstly, it was identified that monitoring of serverless edge computing functions needs to be separated into three perspectives, as summarized here:

- From bottom up, the first identified perspective is monitoring from end-devices: Serverless functions can be monitored by end-users and end-devices. This perspective typically sees serverless functions as so called black boxes and does not differentiate between serverless functions or any other network service. What matters most is the time (end-to-end latency) and used bandwidth to get the desired results.
- The second identified perspective is the perspective of the serverless function developers. As stated by Yan et. al "*Serverless abstracts away the notion of a server from the developer, and pushes operational concerns to the providers of the*

serverless runtime. Despite this, it will be important for developers to monitor the deployment of their solutions since the execution of their functions is directly mapped to a cost model that charges for execution time (typically) and not for idle time" [YCCI16].

- The third perspective is meant to deal with monitoring of the underlying infrastructure of serverless edge computing functions. This is not a necessity for serverless developers in FaaS environments but is necessary for both the FaaS vendors and the serverless edge computing developers because, as identified in this thesis, all serverless edge computing frameworks are currently only available for self-hosting.

Section 4.3.3 lists an analysis about monitoring solutions for AWS Lambda. The finding is that there are three common solutions: The integrated monitoring tools of serverless frameworks like e.g. AWS Cloudwatch; services like IOpipe whom add additional code wrappers to each serverless function that sends information to external services during each function invocation - this increases function latency; and services like Thundra or Dashbird who load metrics from AWS Cloudwatch and enrich those metrics with additional procedures - those services usually also support a *warm-up service* which allows sending heart-beat-like requests to serverless functions to keep the internal AWS Lambda containers running and to prevent cold-starts.

In Section 4.3.4 it was analyzed how to monitor AWS Greengrass. The main result of the available online material is, that there is no enhanced edge monitoring as there exists with AWS Cloudwatch for AWS Lambda. AWS Greengrass monitoring does not provide much information about the performance and resource use of serverless functions. The author of this thesis found, that the most information can be, legally, parsed out of AWS Greengrass log files. Other sources like AWS Greengrass internal IPC-socket could also be used but this is prohibited by AWS License agreements and was therefore not pursued further. Because there is no proper documentation about AWS Greengrass log-file structure, Section 4.3.4 can be seen as a contribution. The different log files and their purposes are listed there. Furthermore, details about the logging of serverless edge computing function invocations are given there. Another result that is based on research about AWS Greengrass is, that there is no built-in way to capture the system utilization of AWS Greengrass devices and, beside RAM, no way to capture the utilization of each function invocation (e.g. CPU-usage per function). Due to license agreements of AWS that prohibit changes to AWS Greengrass Core source code, the only way to add this functionality would be external logging services that run in parallel to AWS Greengrass on the same hardware. But those additional services would again use system resources themselves and furthermore, there is no best practice to combine external measurements about the system utilization with the logged processing time of function runs. Another finding is, that capturing the current system utilization is also possible from within a serverless Greengrass function. This is similar to cloud-based services like IOPipe, but this is limited in practice because doing that increases the processing time of the affected function slightly and if overloading occurs and the queue of triggers runs over,

not all triggers will actually be executed in a serverless function - therefore no monitoring happens.

Because of those findings, a prototype was developed. The aim of the prototype was showing that the log entries of AWS Greengrass, that indicate when a function gets invoked and when results are returned to the invoker, fit the request (end-to-end) latency that is experienced by a client-device that connects to AWS Greengrass. The result of this experiment indicate that the logs of AWS Greengrass are actually not sufficient! Function triggers seem to be queued internally and the log files only contain information about the actual execution of the functions - when they are taken from the queue to be processed, no logging about adding requests to the queue. Benchmarking AWS Greengrass from client devices shows that the execution duration (request latency) that is logged by AWS Greengrass is similar to the processing duration that can be captured inside of the AWS Lambda Greengrass function. But both measurements are far off the experienced request (end-to-end) latency at the AWS IoT client devices. Network problems are very unlikely here because the client device was directly connected to the Raspberry Pi. More likely, the internal MQTT server is queuing the requests when AWS Greengrass' Python-Runtime implementation is not consuming requests in time. The following paragraph describes the implementation of the sample serverless function that was used to find this results and summarizes the results of the experiment with monitoring AWS Greengrass.

5.3.2 Sample Workload Implementation

In order to test the performance of a serverless function, of course, a serverless function is needed as an example. To demonstrate the concepts with realistic edge analytic workload, this function was developed with Python and OpenCV. The function uses OpenCV to detect how many human faces are contained in an image that is sent to the function as payload of the MQTT-trigger. Scikit-learn was also used together with the famous MNIST dataset first, but it quickly turned out that detecting faces with OpenCV is computationally much less intensive than detecting handwritten digits with Scikit-learn on a Raspberry Pi 3 and that the Face Recognition use case is therefore much more suitable for further experimentation. Listing 5.1 shows the python code of the serverless function that was used for this purpose.

Listing 5.1: Face Detection with OpenCV2 in a AWS Lambda Function written in Python 2.7

```
import json
import greengrasssdk
import numpy as np
import cv2
import time
import base64

print("Starting face_detection_lambda_handler python file")
```

```
###
### Global Variables for Configuration

# Lambda (pythonic, not serverless) for getting the current
   ↪ system time.
CURRENT_MILLI_TIME = lambda: int(round(time.time() * 1000))

# OpenCV's config for detecting faces
FACE_CLASSIFIER = cv2.CascadeClassifier('
   ↪ haarcascade_frontalface_alt2.xml')
SCALE_FACTOR = 1.9
BLUE_COLOR = (255, 0, 0)
MIN_NEIGHBORS = 1

# AWS Greengrass relevant configuration
awsiotclient = greengrasssdk.client('iot-data') # used to send
   ↪ mqtt messages to Greengrass' MQTT broker

###
### AWS Lambda handler for processing the image request
###
def face_detection_lambda_handler(event, context):

      # to calculate the processing time (inside of this Lambda
         ↪  function context) the starting time is assigned to
         ↪ a variable
      timeStart = CURRENT_MILLI_TIME()

      # Input parsing and decoding of the base64 encoded image
      imagebase64 = event['imagebase64']
      np_decoded_arr = np.fromstring(imagebase64.decode('base64'
         ↪ ), np.uint8)
      img = cv2.imdecode(np_decoded_arr, cv2.IMREAD_COLOR)
      x = int(event['sizeX'])
      y = int(event['sizeY'])
      xOff = (x-200)/2
      yOff = (y-200)/2
      if xOff != 200 or yOff != 200 :
            crop_img = img[xOff:xOff+200, yOff:yOff+200]
      else :
            crop_img = img
      gray = cv2.cvtColor(crop_img,cv2.COLOR_BGR2GRAY)
```

```
    # Running OpenCV to detect faces in the image:
    faces = FACE_CLASSIFIER.detectMultiScale(gray,
        ↪ SCALE_FACTOR, MIN_NEIGHBORS)

    # Building reply message:
    message = {}
    message['nrOfFaces'] = len(faces)

    # Processing Time that was measured inside of this AWS
        ↪ Lambda function
    message['processingTime'] = str(CURRENT_MILLI_TIME() -
        ↪ timeStart)
    message['invocationId'] = str(context.aws_request_id)

    # Send the message to the topic that is specified as the
        ↪ desired reply address
    awsiotclient.publish(
          topic='rq3/edge/faces/reply'+event['returnTo'] ,
          qos=0,
          payload=json.dumps(message)
    )

    return {
          'statusCode': 200,
    }
```

This serverless function gets triggered if a MQTT message from an AWS IoT device is received on its configured topic on the internal message broker of AWS Greengrass. Connected to this broker, IoT-devices send MQTT requests to AWS Greengrass. Those requests contain encoded images which contain multiple persons' faces. The serverless function takes this image as an input and uses OpenCVs built-in facial recognition capability to count the faces in the image. This count will be returned to a parameterized reply topic via MQTT where the client is listening and waiting for its reply.

5.3.3 Resulting discrepancy in AWS Greengrass Logs

Image 5.1 shows 5 different aggregated measurements of one benchmark. One of them is the *Lambda Processing Time* that was measured inside of the Lambda function. It was measured by remembering the current system time at the start of the Lambda functions' handler and before the request was, via MQTT, sent back to the invoking AWS IoT client.

The variable *Greengrass Processing Time* represents the time that AWS Greengrass

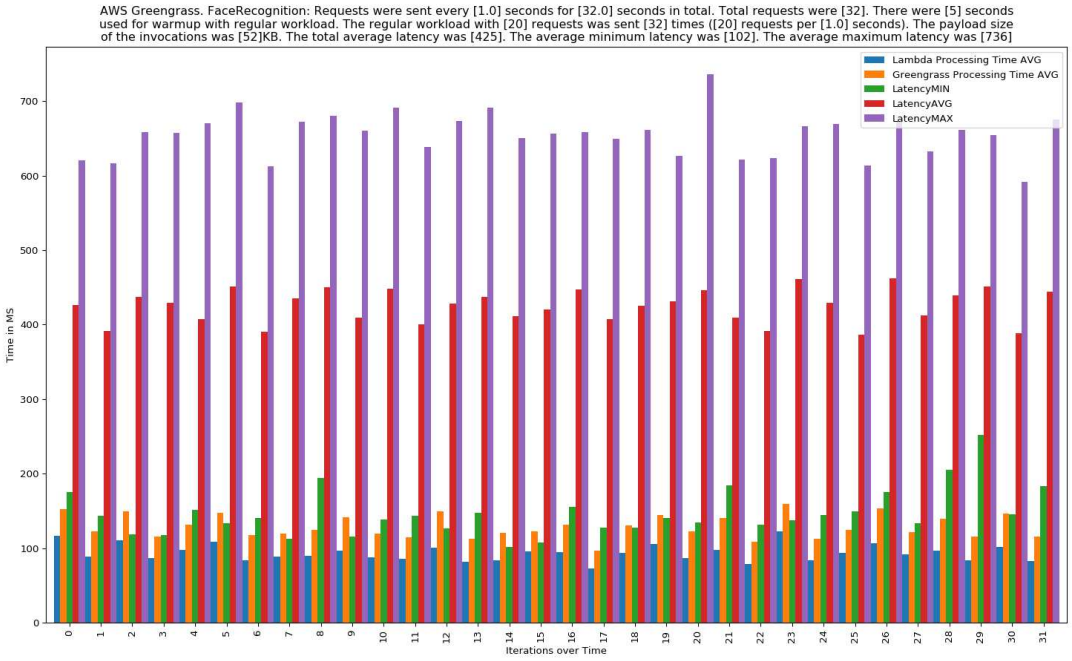


Figure 5.1: Aggregated measurements for a experiment where 20 requests per second being sent to AWS Greengrass for 32 seconds in total where every request had 52KB of payload.

persisted in its log files as time between a work item was received and the answer was posted to MQTT. It can be seen that the Greengrass Processing Time is always larger than the Lambda Processing Time, which makes sense as the Lambda processing time was measured inside of the Lambda function while the Greengrass measurements were done by the serverless framework. The other measurements in the picture show the average, the minimal and the maximum request end-to-end latency for every batch that was measured on the client device (that was sending requests to the AWS Greengrass edge via MQTT). The experiment only took 32 seconds while every second 20 images where sent by the network clients to the AWS Greengrass node because this short execution time already shows the general AWS Greengrass behavior of not logging queued requests.

Image 5.2 shows a totally different structure as the previous test run. The difference between this experiment and the previous one is that here, 50 requests were sent per second for 32 seconds in total. Previously it were only 20 requests per second. The image clearly indicates that AWS Greengrass can not reply to the requests in time and therefore starts to lag behind. It would be expected that the increasing request end-to-end latency (how long it takes until the client device receives and answer for its request) would also be indicated through increasing processing time or at least increasing time that flys

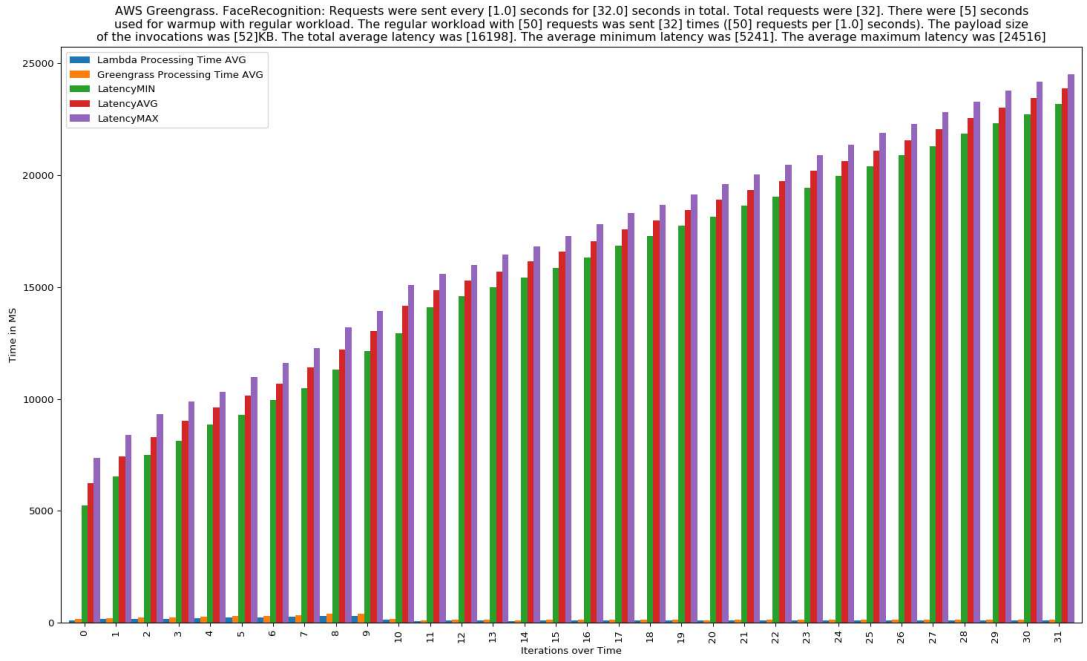


Figure 5.2: Aggregated measurements for a experiment where 50 requests per second being sent to AWS Greengrass for 32 seconds in total where every request had 52KB of payload.

by between AWS Greengrass receives a work item and the time the requests are done. But surprisingly this is not the case. As it can be seen in the Image, the Greengrass Processing Time is not increasing. There is no indication of overloading based on the log files of AWS Greengrass! AWS does not log a timestamp when MQTT requests are received. It only logs when those MQTT requests are processed by the serverless function handler. There is no information about how long the message was inside of the queue.

To summarize this findings and bring them in context to RQ1.3 *How can serverless edge computing devices and the corresponding function runs be monitored and stored (e.g. in AWS Greengrass)? Which tools are required?* the following can be said: AWS Greengrass' built in monitoring tools are not sufficient for monitoring the performance of serverless edge computing functions. It was found that the problem can be solved from three different perspectives. One of them is monitoring on client devices, which was shown to work fine but is limited to demonstration setups. In real-world setups, client devices should not be responsible for monitoring the edge computing nodes. Another perspective comes from inside of the serverless functions where the developers of those functions can access the current system utilization with various python tools and can also persist their execution time. But accessing the used resources of the individual

function run is not possible. Furthermore it brings some overhead to do this inside of serverless functions and especially in FaaS environments, developers tend to reduce the runtime duration of their functions because runtime duration is directly mapped to costs there. The third monitoring perspective comes from the serverless framework, in our case AWS Greengrass. The framework should, ideally, persist as much information as possible to allow developers and the on-premise operator to analyze and act on various system states. Unfortunately, AWS Greengrass built-in monitoring does not meet this objective. It has to be stated that AWS Greengrass monitoring can be configured for each AWS Greengrass Group in the AWS Console and was set to the most fine-grained, detailed setting throughout the analysis in Section 4.3.4 and the benchmarks in this Section. Still, it was shown that the logs of AWS Greengrass do not show any indication that there are already hundreds of requests queued. It was shown by monitoring the increasing response time on client devices that Greengrass is not aware of being overloaded.

5.4 RQ 2 - Machine Learning Results

RQ2: In which ways can a serverless edge computing framework decide where to execute a function (local or cloud) given the current utilization (CPU, RAM, bandwidth, availability of a GPU), QoS (latency, processing time, metrics from RQ1.2)?

Going into detail about Research Question 2 it first has to be stated, that this thesis found out that there is actually no proper way to combine the utilization monitoring measurements done on an AWS Greengrass core-node with the performance measurements of a given serverless function invocation on AWS Greengrass because monitoring of AWS Greengrass does not return any indication for the end-to-end latency of client devices. This result was already discussed in Section 5.3.

This section lists the results of an online and literature research that was carried out for this thesis where relevant topics about machine learning on edge devices were researched. This was initially done to find suitable ways to train a predictor for predicting the execution time of a serverless function but as there is not sufficient training data this was not done. During the literature analysis and by combining the results of multiple related work, it was identified that traditional machine learning approaches as batch learning are a bad fit for serverless functions. Moreover, online-learning, incremental learning, Hypothesis Transfer Learning and Federated Learning were identified as a possible solution for efficient edge analytics in Section 3. This section, here, presents more findings about that. Chapter 6 then discusses those findings together with the other results.

During an comprehensive online analysis about current edge computing hardware and machine learning tools for edge devices, multiple results about practicing machine learning on edge devices were found in Section 4.4.4. One of the main findings is, that on edge devices it has to be distinguished strictly between machine learning training and machine learning inferencing. This is due to various incompatibilities of hardware and software

that will be gone into details in the next paragraphs. Inferencing is, with todays tools, already possible at serverless edge computing nodes but training is still not fully realized.

Another contribution of this thesis is an extensive analysis on the current available serverless cloud providers and serverless edge computing frameworks. This analysis can be found in Sections 4.1.3 and 2.2.5. The analysis was also part of Research Question 1.1. During the analysis about available solutions, one of the main findings was that there is currently no FaaS (cloud-based) service that allows serverless functions to access machine learning accelerators or GPUs. All of the found services rely exclusively on traditional CPU processing. This results in the finding that machine learning will be servery limited in performance. Scikit-learn is a popular entry-level framework for machine learning in Python and does itself only support to be executed on CPUs. But the makers state that recent advances in machine learning like Deep Neural Networks severely rely on GPUs [SL17b]. As the next paragraphs state, Scikit-learn was identified as an easy solution for using machine learning within serverless functions as it fits into the restrictions of limited deployment size and limited execution time of serverless functions quite well. To summarize: Scikit-learn is a nice entry-level machine learning framework, that was shown to work, as well as OpenCV, within AWS Lambda functions on the edge and at the cloud for inferencing as well as training.

Another finding of this thesis is that serverless functions are today not enough to be used in a meaningful way with machine learning frameworks. In fact, it was found that big cloud service providers like AWS, Microsoft and IBM support analytic and machine learning tasks with their dedicated services like AWS Sagemaker and suggest to invoke those *external* services from within serverless functions. Beside the guess that this is due to business reasons, to sell and bind a customer to as much unique vendor-specific services as possible, there are also some technical reasons that were identified in this thesis. One of those reasons is that serverless functions have many constraints that limit their possibility to actually be used for machine learning. This includes a very limited maximum execution time as well as limits for the maximum file size of the function deployment packages. Further details about that can be found in Section 2.2. Scikit-learn was identified as an entry-level machine learning framework that seems to be suitable for building prototypes with serverless functions. The author ran various experiments with Scikit-learn and identified that is is valuable due to its online-training algorithms like Stochastic Gradient Descent and Multi Layer Perceptron Classifiers and Regressors. Scikit-learn is handy to use within serverless functions because it can be compressed to a size of around 48MB which fits into an AWS Lambda function. Furthermore, due to online training, models can be trained while a serverless function is running. If the serverless function terminates (for example because the maximum execution time, 15 minutes at the cloud based AWS Lambda FaaS) the model can be saved to an external service like S3 or a local resource at AWS Greengrass. The next invocation can then retrain the model. Unfortunately, Scikit-learn doesn't support deep learning and also not supports Federated Learning. Implementing Federated Learning with Scikit-learn was tried but resulted in a tremendous amount of additional workload that is not justified

for this thesis. Especially on the long run, Scikit-learn will not support GPU and therefore fast and efficient training of state of the art machine learning models will not be available anyways. The team of TensorFlow is currently working on a early state Federated Learning sub-project which might in future also be available for ARM based edge computing devices.

Another reason why serverless edge computing is limited is the lack of available hardware. Similar to the mentioned cloud-based serverless functions, edge based serverless functions suffer from missing support for hardware-accelerated machine learning. As part of this thesis, a hardware study about the available edge computing hardware was done in Section 4.4.4. Unfortunately, it turned out that most of the available hardware is only usable for machine learning inference and not for machine learning training. While this is fine for using analytics at the edge or for preprocessing data, it does not reduce the required bandwidth to a minimum because data still needs to be sent to the cloud to be used for machine learning training there.

Limited hardware is not the only reason why serverless edge computing has proven to not be usable for many machine learning tasks. There are various serverless edge computing frameworks and the mayor ones were discussed in Section 4.1.3. None of them supports machine learning training out of the box. Some of them, AWS Greengrass, Microsoft IoT Edge, Floggo, support inferencing of machine learning models that were trained somewhere else. But there is currently no framework that can already use data, that is available at the edge, for model training. The only existing solution is, as already mentioned above, to invoke external MLaaS like AWS Sagemaker from within serverless functions. But this means that the training code will not be executed in a serverless function.

The literature analysis in Chapter 3 has shown that Federated Learning is a promising approach for training machine learning models on non-idd data in an unstable and geo-distributed setting. Based on this finding, an evaluation about bringing this approach to serverless edge computing was done. Unfortunately, current implementations of Federated Learning do not support ARM based hardware (which is, as it was found in the hardware study, widely used in edge computing). Furthermore, Federated Learning is a complex research field in an early stage. But still, there are many indications, that specially for the use case, that was discussed throughout this thesis - scheduling serverless functions on the edge or the cloud, based on current utilization - Federated Learning is probably not bringing much advantages. This is, due to the very small training data size. Beside the matter of fact that Research Question 1.2 and 1.3 found that the required training data can not be obtained from AWS Greengrass, it got clear that this training data samples are of very small file size and could probably be transferred to the cloud quite easily. This is due to the fact, that Federated Learning averages the machine learning model weights between the different participating learners (e.g. edge devices). As soon as the synchronization and averaging of models uses more bandwidth as sending the raw training data to a cloud backend, Federated Learning is not worth its computational workload at edge devices anymore. This would for example not be the case for rich media

data - which has much larger file size - but might be the case for predicting the execution time of a serverless function based on system utilization - typically a few numbers per function invocation. For use cases where transferring training data to the cloud is limited due to privacy restrictions, Federated Learning might be a valuable option - but this is then not due to common challenges in edge computing like bandwidth-restrictions. Chapter 6 is used to discuss the results of this thesis in the context of bandwidth, latency and privacy in more detail.

5.5 RQ 3 - Forwarding Results

To what extent does edge analytics improve the latency and overall processing time of serverless (edge) functions? In which ways does the suggested approach (for selecting the destination of a function) change the performance of the function. Discussion of the results is required.

The comprehensive online analysis has shown that AWS Greengrass can be considered as the most mature serverless edge computing framework by answering Research Question 1.1. Later on, it was shown by answering Research Question 1.3 that AWS Greengrass lacks sufficient monitoring capabilities for the performance of serverless functions. Therefore, it was impossible to build a machine learning model that decides on forwarding requests from AWS Greengrass to AWS Lambda in case of overloading because, simply, there is no data available.

Still, this section presents many insights about AWS Greengrass and presents the surprising result that the initial hypothesis, that forwarding requests to the cloud if the edge is overloaded could improve end-to-end latency, actually is wrong. It was shown that forwarding requests from AWS Greengrass to AWS Lambda is tremendously slow, much slower than sending requests to the cloud from IoT devices directly. It was shown that it is faster to process requests for an analytic serverless function (inferencing) on the edge if the requests contain big payload. If the payload is small, it was still shown that its faster to use AWS Greengrass until the host of AWS Greengrass (a Raspberry Pi 3) starts overloading computationally. If overloading occurs it was shown to be faster to send (the small) requests to the cloud because with smaller payload and the same internet uplink, as it was used for the experiment with big files before, bandwidth is not limiting the cloud-based AWS Lambda ingest anymore.

To summarize: It was shown that overloading can make AWS Greengrass slower than AWS Lambda when the requests are small and the internet uplink is strong enough. To the contrary, AWS Greengrass turns out to be faster than the cloud, even when AWS Greengrass is overloaded, if the request payload size exceeds a certain threshold (which obviously depends on the available internet uplink). This means that it was found that forwarding would not improve the end-to-end latency, even if the AWS Greengrass implementation would be better, if the sent payload is too big. Then bandwidth limits the whole use case.

5.5.1 Benchmarking Procedure

In order to structure the experiments, a benchmarking procedure was needed. In general, every experiment was done on simulated AWS IoT devices that were sending requests to different endpoints. There are generally three different types of endpoints: AWS Greengrass, AWS Greengrass (forwarding mode) and AWS Lambda via AWS IoT Cloud.

Workload In order to benchmark a serverless function on AWS Greengrass, it was required to develop a serverless function that runs on AWS Lambda and AWS Greengrass. For that, again an image recognition function is used. The code for AWS Greengrass is exactly the same as in Code Listing 4.1. The code for AWS Lambda is mostly the same and only misses Greengrass specific code blocks, therefore, to save some paper, another code listing is omitted here. Beside minor changes between AWS Greengrass and AWS Lambda, the OpenCV dependencies are different in the deployment packages for AWS Lambda and AWS Greengrass. This is, because of the different hardware architectures that are used for serverless functions on Greengrass and cloud-based Lambda (ARM vs x64)

Payload Size In order to test different scenarios, the image recognition function was benchmarked with two different sizes of payload: 7KB and 52KB. This was selected because 7KB is already really small for a picture and because an image size of 52KB is already close to AWS Greengrass maximum message size limitation of 128 KB per message [AWS18a]. The message might contain additional overhead beside the actual message payload. Beside that, the next paragraph goes into details about AWS IoT's maximum throughput per second (regardless of the message size, each AWS IoT device is not allowed to send more than 512KB per second) which was identified as a limiting factor during this thesis and that's why 52KB was chosen as an arbitrary, realistic, exemplary payload size.

MQTT and Discovery Requests from the devices are sent via MQTT to Greengrass and to AWS Lambda.

For Discovery, in order to connect to AWS Greengrass, an AWS IoT device needs to lookup the current local IP address of its AWS Greengrass group core. AWS currently only allows a static configuration where each AWS Greengrass group contains exactly one core (the edge device) and up to 200 AWS IoT devices [AWS18a]. This discovery process takes a few seconds on every AWS IoT device. Discovery only works if the IoT devices is connected to an internet uplink because the configuration, current encryption keys and the local ip address of the AWS Greengrass core is only available at the AWS cloud.

For AWS Lambda, AWS IoT devices send their requests directly to an AWS IoT Cloud endpoint. There, AWS IoT cloud forwards the MQTT message to a serverless Lambda function. During the experiments it was evident that connecting AWS IoT devices to AWS' MQTT message broker worked well but the performance of sending requests over a

built-up connection was very slow. The requests seem to be buffered at the client. After tremendous online research and code analysis it was found that each AWS IoT device is limited in the data rate that it is allowed to be sent to AWS IoT Cloud. The maximum throughput per second per connection is 512 KB. To circumvent this limitation, the developed benchmarking tool had to incorporate that each client connection can only be used for (512KB divided by the sent message size) messages per second. To incorporate that, during benchmarks, enough clients were started to not be limited by AWS IoT's maximum per-device throughput.

Forwarding In order to test the forwarding of requests from AWS Greengrass to AWS Lambda, the internal forwarding mechanism of AWS Greengrass is used. At the AWS Console it is possible to configure that received MQTT messages on a certain topic on the internal MQTT broker will be forwarded to another topic at the AWS IoT cloud or locally to another AWS IoT Greengrass device. This procedure is called *Subscriptions* in the AWS Greengrass configuration.

5.5.2 Experiment runs

This section is used to show the results of various test runs from the previously described benchmarking experiments. The results are split up into four groups: Experiments with small payload, experiments with big payload, experiments with bursts and experiments with enabled Greeengrass' MQTT-forwarding.

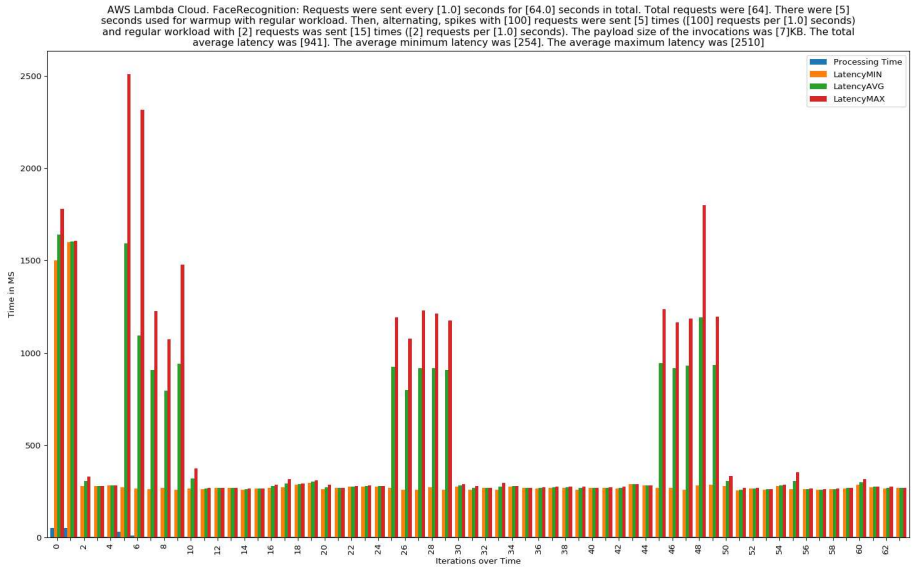
Overloading due to bursts

First, some experiments were undertaken to show that its fine for an edge device to handle some requests per second but that bursts (many requests for a short period of time) quickly floods the device. Figure 5.3a and 5.3b show the measured latency of sending a batch of requests once per second. The minimum, maximum and average request latency show how long the client device (AWS IoT device) had to wait for its reply for the requests sent once per second. The processing time shows how long it took the serverless function to process the requests of every group (per second). As already mentioned in Section 5.3, processing time does not correlate with the request latency that is observed. It seems that AWS Greengrass internally queues requests and does not process them in parallel.

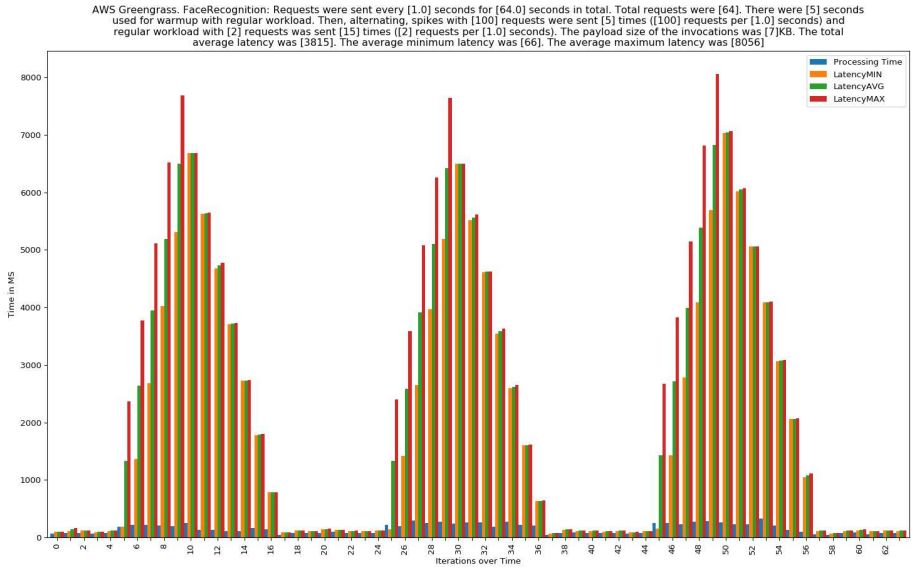
Figure 5.3a nicely shows the cold-start behavior of AWS Lambda. For details about cold and warm starts see Section 2.2.2. The first few requests have a significantly slower response time than the others. This is because before this test run, no requests triggered the AWS Lambda function for some time - internally AWS Lambda probably stopped the container of the serverless function.

To simulate regular workload, 2 requests (to count human faces in an image sent as a payload) were sent once per second for 15 seconds. Afterwards, for 5 seconds, 100 requests were sent per second. Then, again regular workload was sent. Figure 5.3a shows

that the cloud can easily handle the load spikes and that the request latency rises only during the bursts but as soon as the bursts stop, the request latency returns to normal. Figure 5.3b shows the results of the AWS Greengrass edge. There is clear to see that the bursts can not be handled quickly enough. The response time increases. When the bursts stop, AWS Greengrass is still busy for at least another 5 seconds to process all the requests that arrived during the burst. Then, everything returns to normal but it took some time to recover. With the regular workload of only 2 requests per second, the response time is smaller with AWS Greengrass but with bursts, AWS Greengrass quickly gets overloaded. An extreme example for overloading where the edge node never recovers can be seen in Figure 5.2.



(a) Results of experiment with AWS Lambda (cloud-based) with sending small requests (7KB) with 2 requests per second for 15 seconds followed by 5 seconds of 100 requests per second. The total average latency was 941ms.



(b) Results of experiment with AWS Greengrass (edge-based) with sending small requests (7KB) with 2 requests per second for 15 seconds followed by 5 seconds of 100 requests per second. The total average latency was 3815ms.

Figure 5.3: Experiments that shows how bursts with up to 100 requests per second and a payload size of 7KB overload an edge device for significantly longer than AWS Lambda in the cloud

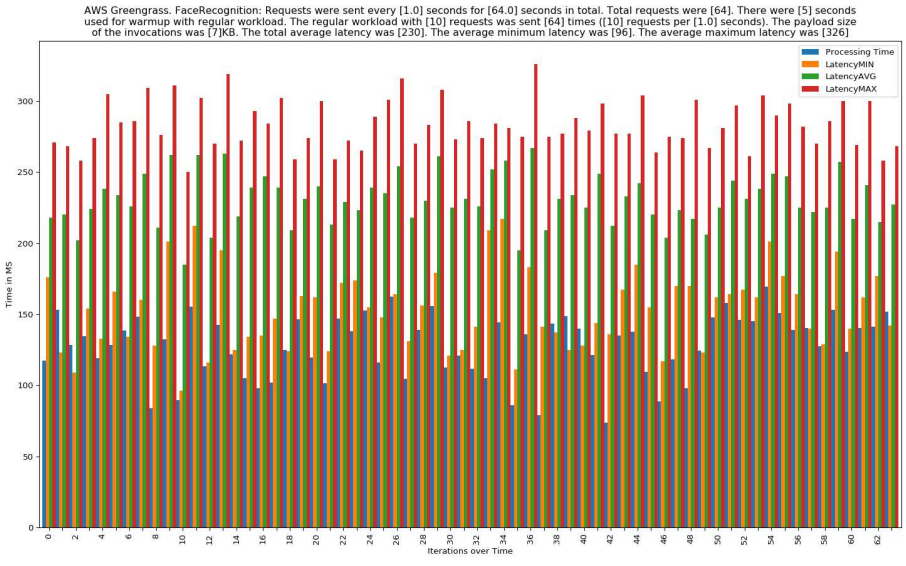
Forwarding slower than processing locally

One of the most important findings is that forwarding requests that are sent to AWS Greengrass to be forwarded to AWS Lambda is tremendously inefficient. That is why experimenting with forwarding was only included into one experiment run. Forwarding is so slow that it is never faster than processing requests exclusively locally or exclusively in the cloud. The reasons for this behavior of AWS Greengrass is unknown and can be part for future research in this topic. Its due to some internal implementation details of AWS Greengrass.

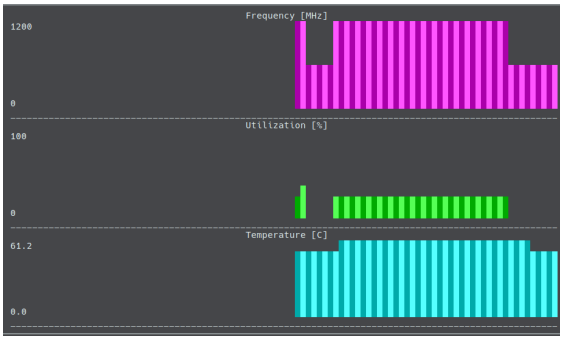
Figure 5.4a, 5.4b and 5.4d show the observations of sending 10 requests per second with an image size of 7 KB per request. There were no burst, every seconds exactly 10 requests were sent.

The results of the experiment show that for 10 request per second with a payload of 7KB each, AWS Greengrass performs best with an average response latency of 230 milliseconds. This is followed by AWS Lambda (when requests are sent directly to the AWS IoT cloud) with an average response latency of 442 milliseconds. When forwarding requests for AWS Greengrass to the AWS cloud, results indicate massive overloading. It seems like AWS Greengrass can not forward the requests fast enough. It should further be mentioned that this is unrelated to the hardware utilization of the AWS Greengrass node. Figure 5.4c and Figure 5.4e show the utilization of the Raspberry Pi during the experiments. It can be seen that processing the requests locally is more computational work than forwarding them. Still, forwarding takes much, much longer than processing them locally or remotely only.

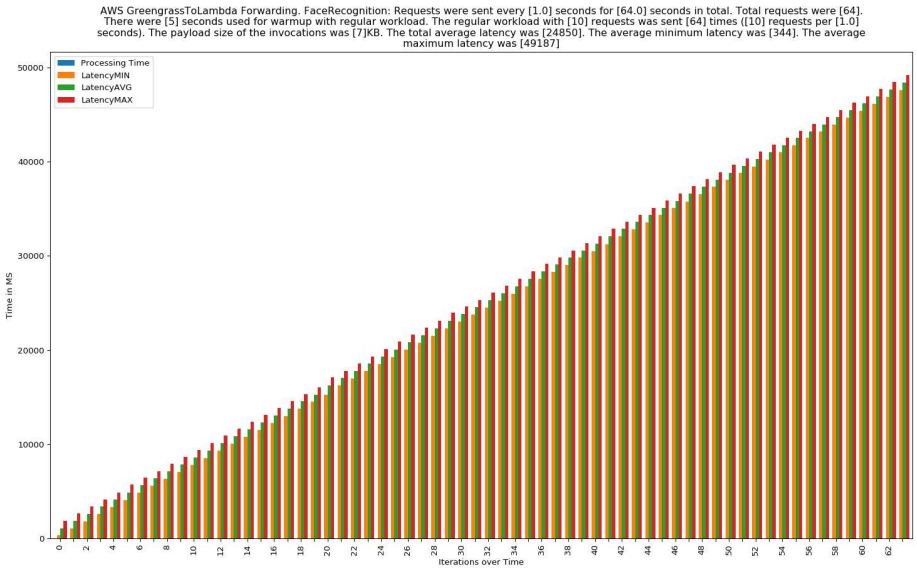
(a) Results of experiment with AWS Lambda (cloud-based) with sending small requests (7KB) with 10 requests per second. The average latency was 442ms.



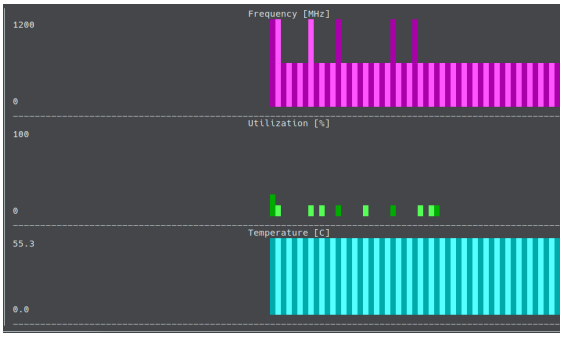
(b) Results of experiment with AWS Greengrass (edge-based) with sending small requests (7KB) with 10 requests per second. The average latency was 230ms.



(c) Utilization of the AWS Greengrass core processing all incoming requests locally without any forwarding. Belongs to Figure 5.4b.



(d) Results of experiment with AWS Greengrass MQTT Forwarding (edge-based) when forwarding all requests to AWS Lambda. Small requests (7KB) were sent 10 requests per second. The average latency was 2480ms but would increase if the experiment would not have been aborted.



(e) Utilization of the AWS Greengrass core while forwarding all incoming requests to the AWS Cloud with AWS built-in forwarding subscription feature. Belongs to Figure 5.4d.

Figure 5.4: Experiment that shows that for 10 request per second with a payload of 7KB each, AWS Greengrass performs best, followed by AWS Lambda (when requests are sent directly to the AWS IoT cloud). Sending requests from client devices to AWS Greengrass and then setting AWS Greengrass to forward all requests to AWS IoT cloud results in massive overloading as it can be seen in the sub-figure 5.4d.

Small Payload

The following experiments result in the following finding: If AWS IoT devices send relatively small requests to AWS Greengrass and AWS Lambda, AWS Greengrass responds faster until around 50 concurrent invocations per second. The experiments were done on a Raspberry Pi 3, different hardware will return different results. With more than 50 requests per second, AWS Lambda is faster. This makes sense because it can not be computationally overloaded. But it was found that with more than 200 concurrent requests per second (with a request size of around 7.5KB per request) the shared internet uplink (2.5MB/s) of the devices begins to struggle. Furthermore, it was shown that the CPU utilization of the Raspberry Pi 3, that ran AWS Greengrass, was at 100% when AWS Greengrass began to overload. This means that based on the CPU utilization, overloading can obviously detected. The following paragraphs present each experiment in detail.

10 Invocations per Second The behavior of sending small payload 10 times per second was already shown above in Figure 5.4a, 5.4b and 5.4d. AWS Greengrass performed much faster than AWS Lambda.

20 Invocations per Second For 20 invocations per second with small payload, results are similar to 10 invocations per seconds. To save ink and because the diagram is very similar to 10 invocations, the diagram is omitted here. AWS Greengrass responds faster than requests to the cloud-based AWS Lambda.

The results for sending the requests from the AWS IoT Devices to the cloud directly are measured once per second: Minimum Latency 254 milliseconds. Average latency 563ms.

Maximum Latency 2029ms.

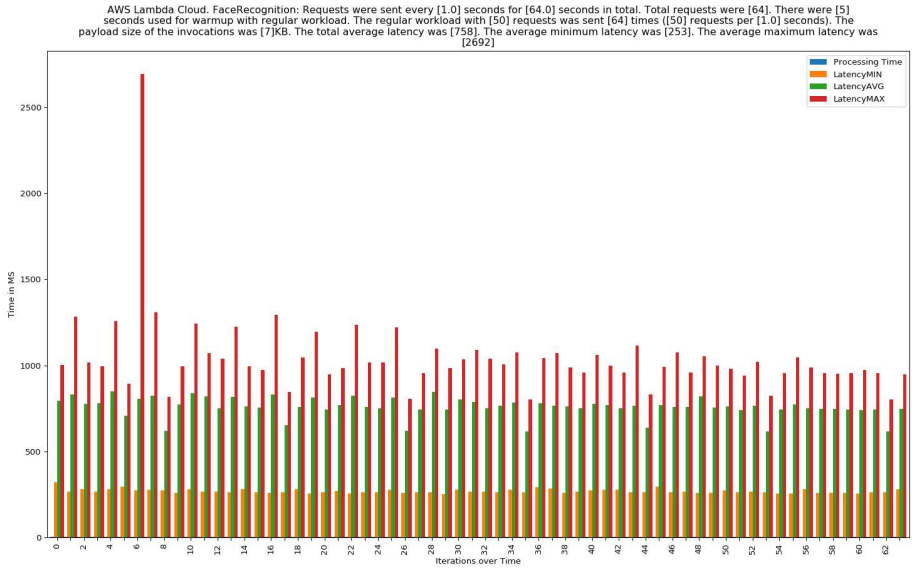
The results for sending the requests from the AWS IoT Devices to AWS Greengrass and processing them there are measured once per second: Minimum Latency 70 milliseconds. Average latency 368ms. Maximum Latency 556ms. During the experiment, the CPU utilization of the AWS Greengrass node was measured and was on average at around 25%.

30 Invocations per Second For 30 invocations per second with small payload, results again similar. Therefore, the diagram is again omitted. AWS Greengrass is slower now but still responds slightly faster than requests to the cloud-based AWS Lambda.

The results for sending the requests from the AWS IoT Devices to the cloud directly are measured once per second: Minimum Latency 256 milliseconds. Average latency 646ms. Maximum Latency 2557ms.

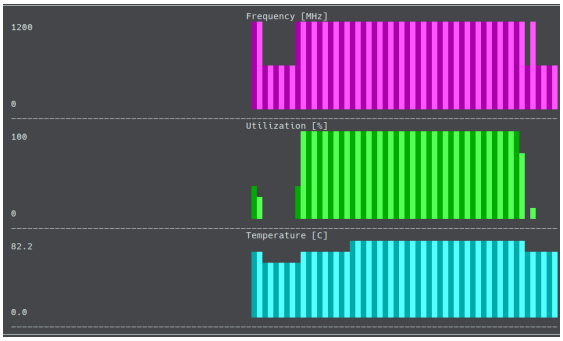
The results for sending the requests from the AWS IoT Devices to AWS Greengrass and processing them there are measured once per second: Minimum Latency 68 milliseconds. Average latency 499ms. Maximum Latency 816ms. During the experiment, the CPU utilization of the AWS Greengrass node was measured and was on average at around 50%.

50 Invocations per Second For 50 invocations per second, the result changed. AWS Greengrass is not able to handle 50 invocations per second with a 7KB image. Figure 5.5b shows how requests are queuing up and take longer and longer to get returned because AWS Greengrass is overloaded. Figure 5.5c shows the utilization of the AWS Greengrass node while its not able to process the requests fast enough. The CPU utilization is at 100%. Figure 5.5a shows that at the meantime, the cloud-based AWS Lambda is on average a bit slower than with 30 requests per second but still manages to return a response on average after 758ms.



(a) Experiment that shows that for 50 request per second with a payload of 7KB each against AWS Lambda. The average latency was 758ms.

(b) Experiment that shows that for 50 request per second with a payload of 7KB each against AWS Greengrass. The average latency was 6624ms.



(c) Utilization of the AWS Greengrass core processing all incoming requests locally without any forwarding. Belongs to Figure 5.5b.

Figure 5.5: Experiment that shows that for 50 request per second with a payload of 7KB each, AWS Greengrass performs badly on a Raspberry Pi 3. The Raspberry Pi 3's CPU is utilized 100% but still, requests are queuing up and response time increases. The cloud-based AWS Lambda (when requests are sent directly to the AWS IoT cloud) performes okay.

100 Invocations per Second For 100 requests per second, AWS Greengrass is of course still overloaded and never recovers. But the cloud-based AWS Lambda works fine. It seems like AWS Lambda works well as long as there is enough upload-bandwidth available for the group of AWS IoT clients. Figure 5.6 shows one experiment with 100 requests per second for AWS Lambda. With an average latency of 993ms, this is still acceptable.

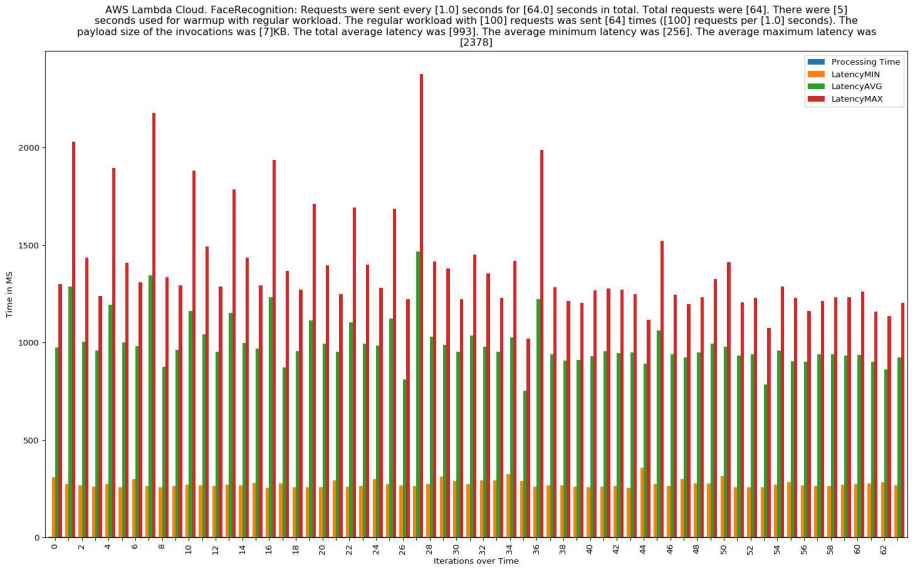


Figure 5.6: Experiment that shows that for 100 request per second with a payload of 7KB each against AWS Lambda. The average latency was 993ms.

200 Invocations per Second For 200 invocations per second, AWS Lambda gets significantly slower. The average latency rises to over 3090ms. This is probably due to the reason that the AWS IoT devices all share the same uplink-internet-connection. This was explicitly built that way to simulate a setup where devices are close (at the same network) as an edge device. Only with such a setup its reasonable to compare the performance of AWS Lambda (directly) with AWS Greengrass. Figure 5.7 shows how the cloud-based approach got slower and more unstable than before.

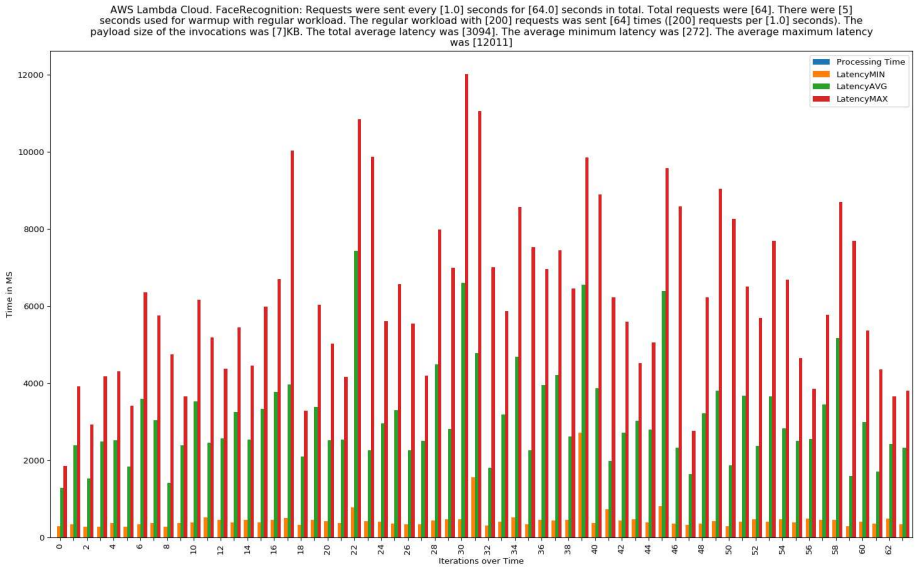
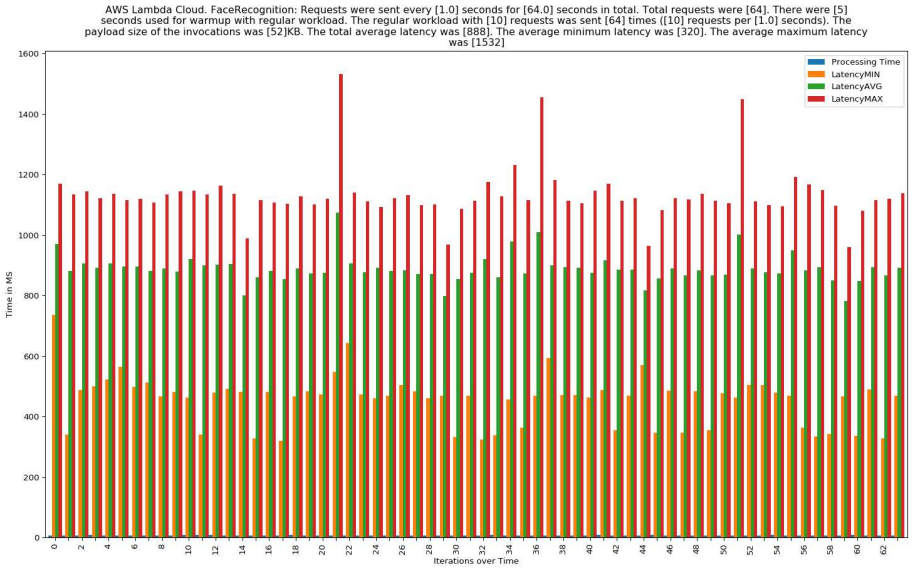


Figure 5.7: Experiment that shows that for 200 request per second with a payload of 7KB each against AWS Lambda. The average latency was 3094ms.

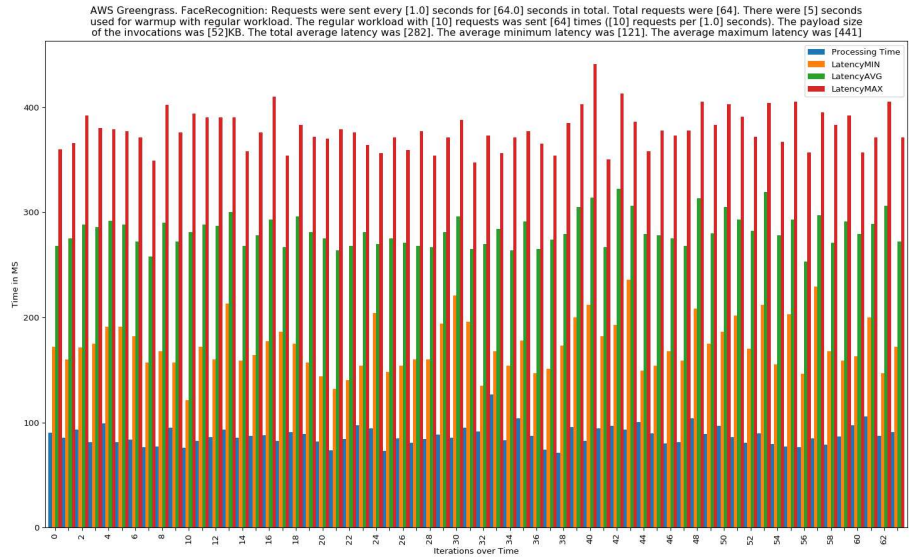
Big Payload

By experimenting with bigger payload, concrete 52KB per request, the following results were obtained and are summarized in the next paragraphs. When sending only few requests per second it was shown that AWS Greengrass takes around 10 times longer to process the request but due to the extremely short network distance and high bandwidth between the AWS IoT devices and the AWS Greengrass edge, AWS Greengrass performs better than AWS Lambda. Even with more requests per second, AWS Greengrass always responds quicker while processing slower. This can be explained by the fact that sending 50 requests per second with 52KB payload each over a shared internet uplink, quickly uses all available bandwidth (2.6MB/s). Its interesting to further mention that the experiments indicate that AWS Greengrass is processing requests First-In-First-Out while AWS Lambda over AWS IoT seems, as it can be seen in the experiments below, to process those requests randomly, sometimes fast, sometimes slow. For large payload it was shown that AWS Greengrass is superior to AWS Lamba. The processing power of the cloud does not help here because the data can not be uploaded fast enough - a typical example for edge computing.

10 Invocations per Second Figure 5.8a and 5.8b show 10 invocations per seconds. Each request contained an image with a size of 52KB. It can be observed that the average response latency is much lower for AWS Greengrass in Figure 5.8b than it is for the cloud-based AWS Lambda in Figure 5.8a. The minimum and maximum observed request latency is also below the cloud endpoint. While the response time was much smaller for the edge device, it can be seen that the actual processing time for each image is somewhere at around 100 milliseconds for the edge-based Lambda function and only at around 10 milliseconds for the cloud based AWS Lambda function. This means that the edge device takes about 10 times longer to process the request. But still, due to network latency, which is mainly influenced by the uploading of the image to the cloud, the edge device returns the reply faster than the cloud.



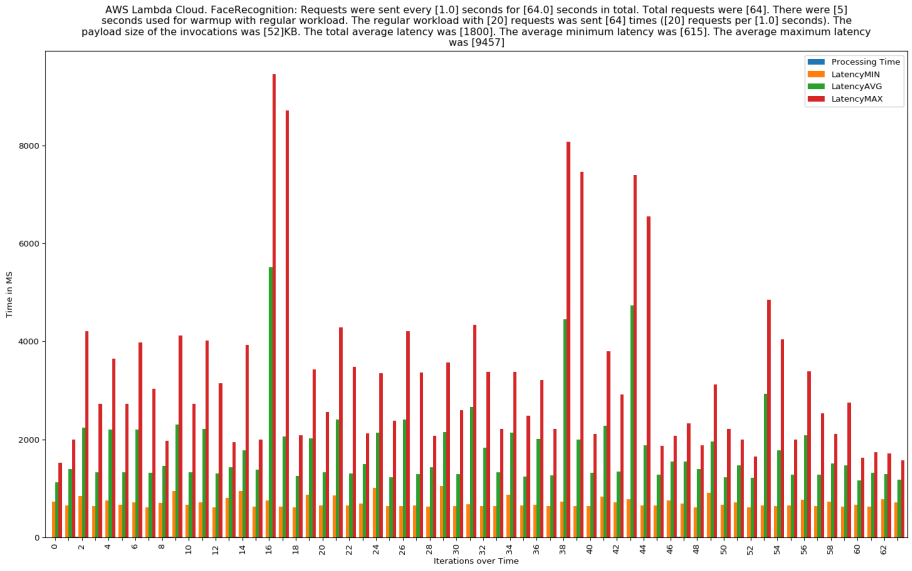
(a) Experiment that shows that for 10 request per second with a payload of 52KB each against AWS Lambda. The average latency was 888ms.



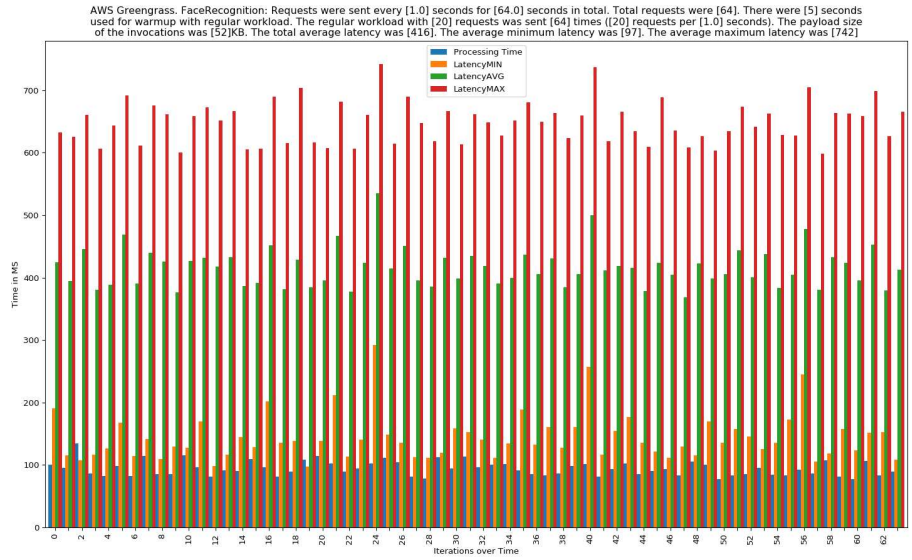
(b) Experiment that shows that for 10 request per second with a payload of 52KB each against AWS Greengrass. The average latency was 282ms.

Figure 5.8: Experiment results for 10 invocations per second with 52KB payload per request

20 Invocations per Second After sending 10 invocations per second, the rate was increased to 20 invocations per second. Here, the results are similar than with 10 per second. AWS Greengrass is able to, with around 450ms on average, respond significantly faster than AWS Lambda, with around 1800ms. Figure 5.9a shows the results for sending requests directly to AWS Lambda while Figure 5.9b shows the results for AWS Greengrass.



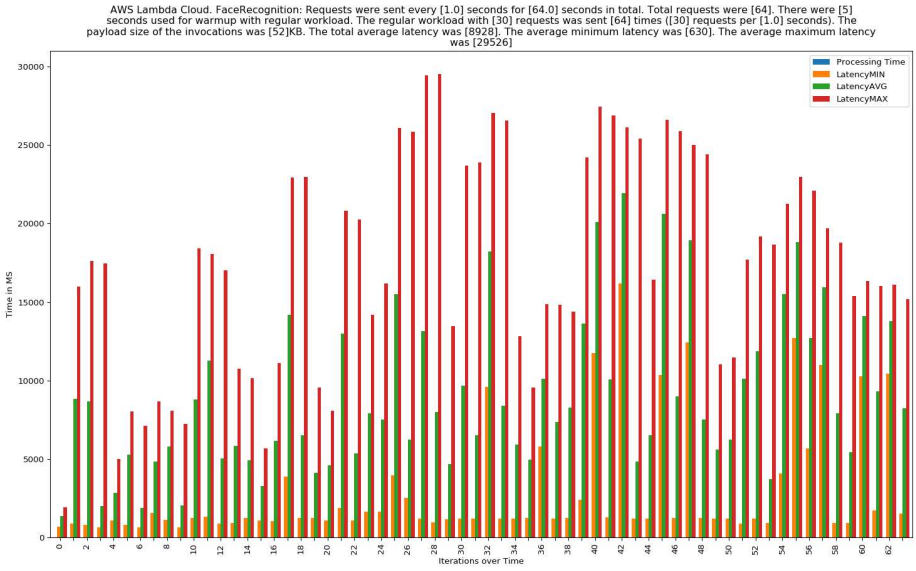
(a) Experiment that shows that for 20 request per second with a payload of 52KB each against AWS Lambda. The average latency was 1800ms.



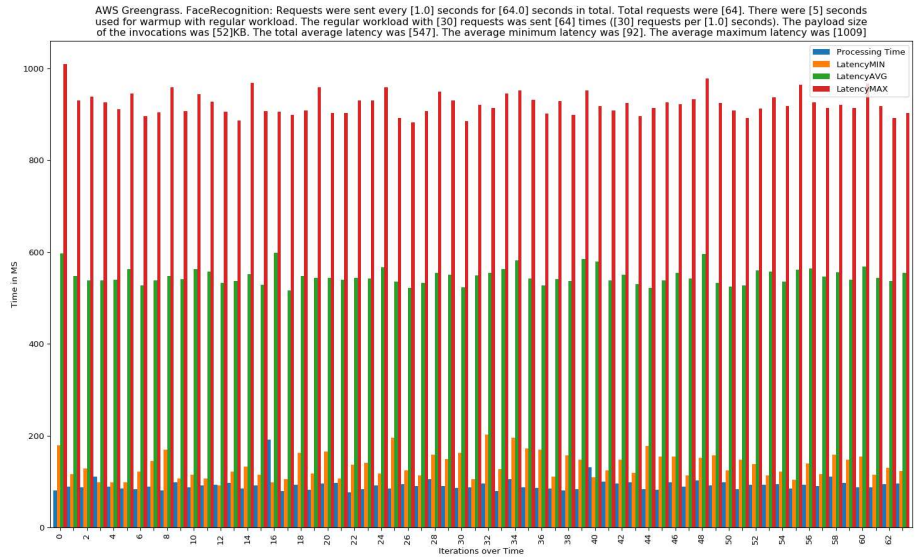
(b) Experiment that shows that for 20 request per second with a payload of 52KB each against AWS Greengrass. The average latency was 416ms.

Figure 5.9: Experiment results for 20 invocations per second with 52KB payload per request

30 Invocations per Second Thirty invocations per second show similar results, AWS Lambda needed 8928ms to respond on average while the processing time was still at around 10ms. Obviously, requests are not transmitted fast enough. AWS Greengras shows similar results as before, 547ms on average are a quite good result for a Raspberry Pi 3. Figure 5.10a and Figure 5.10b show the experiments. Its interesting to look at the results of the AWS Lambda experiment in detail. The digram in Figure 5.10a shows the minimum, the average and the maximum response time for the 30 requests that were sent each second. Surprisingly, the minimum time stays quite constant at around 630ms while the maximum time goes up until 29000ms. 29 seconds are extremely long and in practical consideration not suitable for any application where user interaction is required. The gap between minimum and maximum indicates that AWS Greengrass is not processing the requests in order - like for example first-in-first-out. It does more look like random.



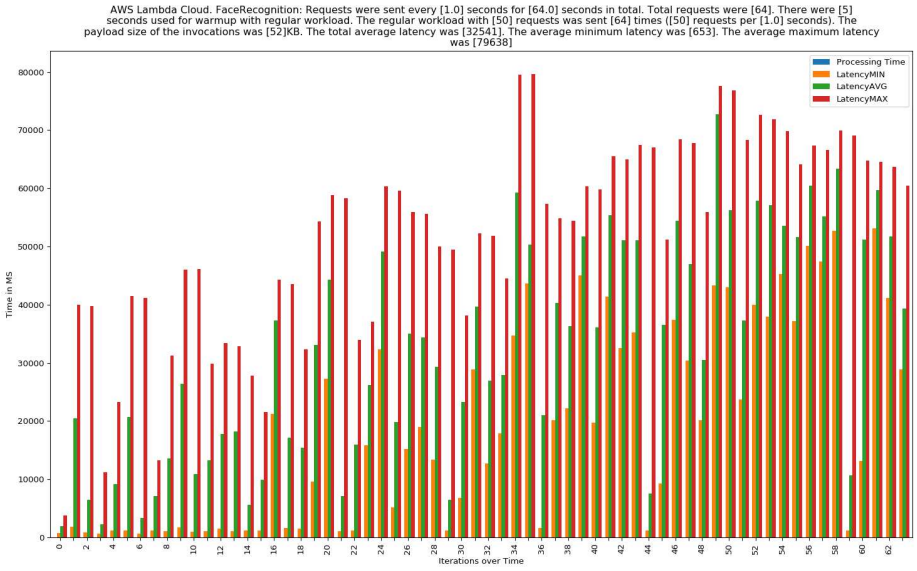
(a) Experiment that shows that for 30 request per second with a payload of 52KB each against AWS Lambda. The average latency was 8923ms.



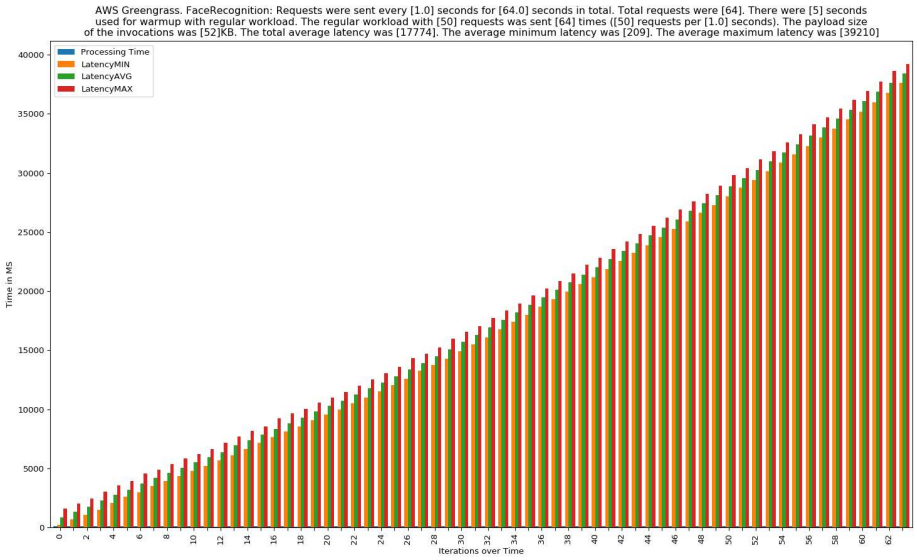
(b) Experiment that shows that for 30 request per second with a payload of 52KB each against AWS Greengrass. The average latency was 547ms.

Figure 5.10: Experiment results for 30 invocations per second with 52KB payload per request

50 Invocations per Second As it can be seen in Figure 5.11b, fifty invocations per seconds with a payload of 52KB clearly overloads the AWS Greengrass edge running on a Raspberry Pi 3. Figure 5.11a shows similar results for AWS Lambda, while overloading does not happen with such a nice pattern, the average response time can clearly be seen to increase every second. Again, as already mentioned at the results for 30 requests per second, AWS Lambda seems to not process invocations in order, some requests finish quite quickly while others need more than a minute to finish. It is valuable to know that AWS Greengrass is strictly processing requests according to the First-In-First-Out pattern. This can clearly be seen in Figure 5.11b. To summarize: AWS Greengrass is computationally overloading and therefore queuing the requests while AWS Lambda receives the requests unordered, probably due to network issues as the sent payload is bigger than the available bandwidth.



(a) Experiment that shows that for 50 request per second with a payload of 52KB each against AWS Lambda. The average latency was 32541ms.



(b) Experiment that shows that for 50 request per second with a payload of 52KB each against AWS Greengrass. The average latency was 17774ms.

Figure 5.11: Experiment results for 50 invocations per second with 52KB payload per request

CHAPTER 6

Discussion

During the progress of this thesis, the main assumption was that serverless edge computing would always improve a cloud computing setup. This chapter shows the implications of the findings from Section 5 and discusses them along four different perspectives. Not all perspectives indicate that serverless edge computing is actually the right fit. Privacy, Latency, and Bandwidth are said to be positively influenced by edge computing but serverless edge analytics was shown to not satisfy them by all means. Furthermore, computational complexity is a perspective where edge omputing and serverless functions generally are yet very limited.

6.1 Use Case 1: Bandwidth

Edge Computing is an enabler for data analysis of data that can, due to bandwidth restrictions, not be sent to the cloud otherwise. The results of this thesis indicate that there is only a small gap between an edge device that is computationally overloaded and an internet connection that is struggling with data throughput. Especially the finding that the cloud-based AWS IoT is limited in data ingest, with only a few KB per second for every MQTT device, shows that AWS actually wants to prevent having high bandwidth data input from IoT devices. Serverless Edge Computing could solve this problem by doing data pre-processing, compressing or data caching in case of load spikes. Bringing serverless computing to edge devices reduces the complexity of deploying edge systems. The only drawback that arises, and this is also discussed in the next use case, is that serverless functions come with slower startup times than traditional services. Especially for applications where latency is critical, pre-processing might take longer than using tremendous amounts of bandwidth to directly ingest data into cloud data centers. This is indicated by the results of Research Question 3 in Section 5.5.

Federated Learning, done on edge computing devices, can also reduce the required uplink-bandwidth. Especially for use cases where the file size of training data samples is large,

Federated Learning will be a good option. For the use cases observed during the study of this thesis, where the training data is very small, like predicting the processing time of a serverless function based on system utilization, Federated Learning might not always save much bandwidth because the synchronization of the machine learning models also requires a certain amount of bandwidth.

6.2 Use Case 2: Latency & offline availability

Edge Computing is said to reduce the latency of requests because the network distance each request has to travel is reduced. Placing edge devices close to end devices of course reduces the network latency. But, while serverless edge computing is reducing the complexity of edge computing deployments, latency is not improved automatically. When a triggers occurs (for example via MQTT), serverless functions are started by the serverless framework. Sometimes the serverless function instances can be reused for other requests again. This behavior is called warm-start but cold-starts do also exist [See Section 2.2.2]. Cold starts can always occur and have dramatic impact on the end-to-end request latency for end devices. In the experiments of this thesis, AWS Lambda had an observed cold-start latency of around three seconds. AWS Greengrass had an observed cold start latency of one to three seconds. Furthermore, AWS Greengrass, as well as AWS IoT with AWS Lambda, queue requests for serverless functions when the computing can not yet happen. While this makes sense it makes complying with Service Level Agreements difficult as latency can rise due to queued requests. Beside that, the discovery process that each AWS IoT device has to execute before being able to communicate with AWS Greengrass also takes up to a few seconds. But this only needs to be done when the IoT device connects to the edge for the first time (or after the edge changes its configuration, certificates, ip-address).

After the found insights of this master thesis, it seems that serverless edge computing is a bad fit for applications with demand for low latency. The overhead of starting and stopping the serverless functions definitely has a bad influence on latency. Generally, for low latency, serverless is a bad fit because the main benefit is actually the ability to execute many different serverless functions that are currently in demand. Due to those functions, not being long running, one edge can potentially cover much more functionality. But if a low latency system is needed, the overhead of starting those functions imposes additional overhead. Therefore, for such use cases, traditional edge computing services might be a better fit. Overloading was identified as the most influential problem in edge computing and this thesis shows that overloading can happen in computational form on edge devices and in form of network overloading due to low bandwidth.

6.3 Use Case 3: Computational Complexity

Baresi et al. have already shown that edge devices can easily be overloaded due to their resource limitations [BMG17]. Similar results have been shown in this thesis. Serverless

functions with high hardware requirements seem to be a bad fit for edge computing. It was found that the default edge computing devices of AWS Greengrass and Microsoft IoT Edge, the Raspberry Pi 3, is very limited for serverless edge computing. Load-peaks can easily overload the device and it was found that for requests (triggers) that have less payload, it can be better - in terms of minimal end to end request latency - to send those requests directly to the cloud without sending them to the edge devices ever. As soon as the payload of serverless triggers increases, the internet upload of end devices quickly runs short. Then, serverless edge devices still overload but might give a reply in nearly the same time as an overloaded uplink - at least this was observed in the experiments of this thesis. Experiments quickly showed response times of above 30 to 60 seconds which would be hard to use for any use case where user interaction is required and low latency is necessary.

Generally it can be said, based on the findings of this master thesis, that serverless edge computing only is a valid competitor to cloud-based serverless computing if the edge devices have high computational resources. Low computational resources can only help improving the end-to-end request latency if the internet uplink is very slow. This means that, the faster the internet connection, the better equipped an edge computing node needs to be.

6.4 Use Case 4: Privacy and Operational Complexity

Beside all mentioned technical reasons to use or not use serverless edge computing beside latency, bandwidth and computational power, there is another reason to do so. Privacy can sometimes require data to stay inhouse of a company. For example in the fields of law and medical professionals, data is highly protected. Sending data to cloud data centers is often prohibited. Sending data outside of the EU is difficult since the General Data Protection Regulation. Serverless edge computing could be seen as a bridge to bring cloud technologies to protected environments. Throughout this thesis, Federated Learning was identified as a future enabler for machine learning in edge computing. This was primarily stated because of technical reasons, like expected bandwidth reductions, but McMahan et al. see chances for Federated Learning on private data [MMRyA16]. When Federated Learning training rounds would be done within serverless edge computing functions, privacy would be protected. Due to reduced deployment complexity and edge specific configuration, data science experts that practice Federated Learning would also benefit from the minimal operational complexity serverless edge computing in comparison to traditional edge computing services.

CHAPTER 7

Summary and Future Work

7.1 Summary

In times of tremendous amounts of data being produced at end devices, Edge Computing is increasingly relevant today. The amount of data that is currently generated but not used for analysis can only be used in future, if processing power is placed close to the end devices. Bringing machine learning and analytics to edge computing is therefore a requirement that seems obvious. Serverless computing is known to reduce the complexity of cloud computing systems due to reduced complexity in billing and deployment and scaling. Recently, cloud vendors started to bring serverless functions to edge computing, with the promise to simplify working in such unstable, highly distributed environments. Combining these recent advances brings up the term of serverless edge analytics.

For this reason, in the course of this thesis, the goal was evaluating the suitability of serverless functions and edge computing for machine learning. To do that, a comprehensive background and related work analysis was conducted to find strengths, weaknesses and coherence of the named technologies. For evaluating those findings, AWS Greengrass was benchmarked and evaluated for its suitability of serving serverless functions dynamically. This was done due to the finding, based on related work, that AWS Greengrass is the most mature serverless edge computing today, still, in the progress of this thesis many drawbacks were identified.

The results of this thesis include findings in various fields. First of all, it was shown that, while AWS Lambda functions can in theory be deployed on AWS Lambda (cloud-based) and AWS Greengrass (edge-based) servers, edge and cloud servers are still not compatible due to constraints regarding the processor architecture and deployment package dependencies. Furthermore it was shown that AWS Lambda functions are limited in size and execution time and that machine learning, especially batch training, is not really suitable for serverless functions. The comprehensive background and related

work analysis has shown that currently, no cloud based FaaS vendor provides access to GPUs within serverless functions and that current edge computing hardware lacks support for machine learning training. ML-inferencing has shown to work well with edge devices. The related work analysis also brought up that Federated Learning can solve many challenges in edge computing.

By analyzing the experiments of this thesis it was shown that AWS Greengrass is not usable for forwarding requests to the cloud statically and dynamically because of AWS' slow implementation and throttling within AWS Greengrass. Also it was shown that for big request-payload AWS Greengrass is faster than AWS Lambda. Even when the edge device (in our case a Raspberry Pi 3) overloads and therefore responds slowly, sending requests to the cloud, where processing power is unlimited, over an average Austrian internet uplink of 2.5MB/s, is not faster. Because of that, forwarding would not even reduce the request latency for such a use case. Another experiment has shown that for small payload, this finding changes to the benefit of cloud computing, as then, the same internet uplink can transfer enough requests to the cloud. For this experiment, the cloud was found to be faster as soon as the Raspberry Pi starts overloading. If the Raspberry Pi edge-based gateway is not overloading it was shown that he is, for image recognition with OpenCV in AWS Greengrass, always faster than the AWS Lambda-based cloud, regardless of the size of the request-payload size.

The findings of this thesis align well with various related work. Most importantly, the work of Baresi et al. about overloading edge and fog computing nodes was once more confirmed with an investigation on serverless edge computing with AWS Greengrass [BMG17]. The work of Hellerstein et al has shown that AWS Lambda can for some use cases be much slower than EC2 [HFG+18] and this was also observed during the experiments due to coldstarts. To finally summarize the findings, serverless edge computing has potential for processing non-time-critical data locally if developers expect traffic-spikes that might overload the edge.

In conclusion, this thesis provides a detailed overview over the current landscape of serverless computing, serverless edge analytics and AWS Greengrass and has shown limitations for dynamically executing serverless functions at cloud and edge devices.

7.2 Future Work

As outlined by this thesis, there is a opportunity field for scientific research in the area of geo-distributed machine learning like Federated Learning in the field of Edge Computing. Furthermore, Edge Computing and especially suitable hardware for edge analytics are an emerging field. Beside that, research in the area of performance limitations of current serverless computing frameworks, monitoring serverless functions and the organization of cold-and-warm starts are identified to be important.

To further pursue the research contributions of this thesis, I would recommend continue looking for an efficient serverless edge computing implementation. AWS Greengrass was

shown to be superior to various other tools based on theoretical background, but still, monitoring AWS Greengrass was shown to be not sufficient. Additional research should also be put into solutions and frameworks for efficiently routing requests between edge, fog and cloud devices as it was shown that AWS Greengrass lacks those properties. DNS or discovery based solutions are thinkable. Serverless functions, as soon as they support arm and x64 processors transparently, which is another task for future work, could be used transparently on edge, fog or cloud devices if routing of requests, so called triggers, is more efficient and further optimized. AWS Greengrass was shown to not be efficient in terms of forwarding requests to AWS IoT.

Further work should also be put in developing hardware and software for training machine learning models on edge computing devices because it was found that this could potentially reduce the required bandwidth of various analytical services.

List of Figures

List of Tables

Acronyms

AMQP Advanced Message Queuing Protocol. 68

AWS Amazon Web Services. 11–13, 20, 32, 37, 55, 60, 67, 71, 72, 100

BaaS Backend as a Service. 9, 11

CaaS Container as a Service. 11

CDN Content Delivery Network. 24, 65, 66, 147

CES Consumer Electronics Show. 61

CoAP Constrained Application Protocol. 22, 23, 54

CPS Cyber-Physical Systems. 22, 24

CPU Central Processing Unit. 22, 98, 101

FaaS Function as a Service. xi, xiii, 9, 11–15, 17–19, 21, 38, 58, 59, 61, 63, 79, 80, 82, 85, 86, 109–111, 116–118, 144

GB Gigabyte. 77

GPU Graphics Processing Unit. 10, 18, 19, 27, 37, 38, 59, 60, 82, 98, 101, 118, 144

HTL Hypothesis Transfer Learning. 43–45

HTTP Hyper Text Transfer Protocol. 23, 54, 67

IaaS Infrastructure as a Service. 9, 10, 75, 79, 80, 110

IBM International Business Machines Corporation. 10, 11, 16, 17

IoT Internet of Things. 17, 22–24, 26, 45, 53, 54, 62, 65, 67, 69–71, 96

MB Megabyte. 77

MEC Mobile Edge Computing. 26, 33

ML Machine Learning. 27, 65, 67, 69, 71, 73

MLaaS Machine Learning as a Service. 11, 119

MQTT Message Queuing Telemetry Transport. 17, 22, 23, 54, 67, 68

MS Microsoft. 10–12

NIST National Institute of Standards and Technology. 9, 11, 12

OS Operating System. 68

PaaS Platform as a Service. 9–11

QoS Quality of Service. 96, 98

SaaS Software as a Service. 9–11, 75, 76, 79, 110

TPU Tensor Processing Units. 25, 61

VM Virtual Machine. 33, 38

Bibliography

[ADC10] M. Alhamad, T. Dillon, and E. Chang. Conceptual sla framework for cloud computing. In *4th IEEE International Conference on Digital Ecosystems and Technologies*, pages 606–610, April 2010.

[Afo19] Tejumade Afonja. Kernel Functions. https://towardsdatascience.com/kernel-function-6f1d2be6091, 2019. [accessed online, 04.02.2019].

[AGM+15] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, Fourthquarter 2015.

[AK17] Tractica Aditya Kaul. Artificial Intelligence Processing Moving from Cloud to Edge. https://www.tractica.com/artificial-intelligence/artificial-intelligence-processing-moving-from-cloud-to-edge/, 2017. [Online, 10.09.2018].

[Apa19] Apache. Apache OpenWhisk System Overview. https://github.com/apache/incubator-openwhisk/blob/master/docs/about.md, 2019. [accessed online, 15.03.2019].

[App17] Apple. iPhone 8 Specifications. https://www.apple.com/lae/iphone-8/specs/, 2017. [accessed online, 17.10.2018].

[ARM18a] ARM. Arm Project Trillium Launch News. https://www.arm.com/company/news/2018/02/arm-project-trillium-offers-the-industrys-most-scalable-versatile-ml-compute-platform, 2018. [accessed online, 17.10.2018].

[ARM18b] ARM. Arm Project Trillium Project Page. https://www.arm.com/products/silicon-ip-cpu/machine-learning/project-trillium, 2018. [accessed online, 17.10.2018].

[AWS14] AWS. Aws Lambda Document History. https://docs.aws.amazon.com/lambda/latest/dg/history.html, 2014. [accessed online, 20.10.2018].

[AWS17] AWS. Aws Lambda FAQ. https://aws.amazon.com/lambda/faqs/, 2017. [Online, Retrieved at 9.3.2019].

[AWS18a] AWS. AWS Limits. https://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html, 2018. [accessed online, 20.12.2018].

[AWS18b] AWS. Environment Setup for Greengrass. https://docs.aws.amazon.com/greengrass/latest/developerguide/module1.html, 2018. [accessed online, 16.03.2019].

[AWS18c] AWS. Greengrass FAQ. https://aws.amazon.com/de/greengrass/faqs/, 2018. [Online, Retrieved at 17.11.2018].

[AWS19a] AWS. AWS CloudFront Limits. https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/cloudfront-limits.html#limits-lambda-at-edge, 2019. [accessed online, 5.3.2019].

[AWS19b] AWS. AWS CloudWatch. https://aws.amazon.com/cloudwatch/?nc1=h_ls, 2019. [accessed online, 13.5.2019].

[AWS19c] AWS. AWS EC2 Costs. https://aws.amazon.com/ec2/pricing/on-demand/, 2019. [accessed online, 2.5.2019].

[AWS19d] AWS. Aws greengrass developer guide / supported platforms and requirements. https://docs.aws.amazon.com/greengrass/latest/developerguide/what-is-gg.html#gg-platforms, 2019. [accessed online, 25.3.2019].

[AWS19e] AWS. AWS Greengrass License. https://s3-us-west-2.amazonaws.com/greengrass-release-license/greengrass-license-v1.pdf, 2019. [accessed online, 8.7.2019].

[AWS19f] AWS. Aws greengrass opc-ua developer guide. https://docs.aws.amazon.com/greengrass/latest/developerguide/opcua.html, 2019. [accessed online, 25.3.2019].

[AWS19g] AWS. AWS IoT Greengrass ML Inference. https://aws.amazon.com/greengrass/ml/, 2019. [Online, 13.10.2019].

[AWS19h] AWS. Aws lambda and aws x-ray. https://docs.aws.amazon.com/xray/latest/devguide/xray-services-lambda.html, 2019. [accessed online, 30.06.2019].

[AWS19i] AWS. AWS Lambda CloudWatch Metrics. https://docs.aws.amazon.com/lambda/latest/dg/monitoring-functions-metrics.html, 2019. [accessed online, 01.04.2019].

[AWS19j] AWS. AWS Lambda Developer Guide. https://docs.aws.amazon.com/greengrass/latest/developerguide/lambda-functions.html, 2019. [accessed online, 25.3.2019].

[AWS19k] AWS. AWS Lambda Limits. https://docs.aws.amazon.com/lambda/latest/dg/limits.html, 2019. [accessed online, 5.3.2019].

[AWS19l] AWS. Lambda@edge. https://aws.amazon.com/lambda/edge/, 2019. [accessed online, 5.3.2019].

[Aya14] Eng Mohannad M Ayash. Research methodologies in computer science and information systems. *Retrieved November*, 28:2014, 2014.

[Azu18] Microsoft Azure. Azure Functions 2.0 available in IoT Edge. https://azure.microsoft.com/en-gb/updates/azure-functions-2-0-available-in-iot-edge/, 2018. [accessed online, 17.03.2019].

[Azu19] Microsoft Azure. IoT Hub quotas and throttling. https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-quotas-throttling, 2019. [accessed online, 17.03.2019].

[BI17] Zoran Ivanovic Boris Ivanovic. How to Deploy Deep Learning Models with AWS Lambda and Tensorflow. https://aws.amazon.com/de/blogs/machine-learning/how-to-deploy-deep-learning-models-with-aws-lambda-and-tensorflow/, 2017. [accessed online, 20.12.2018].

[BIK+17] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1175–1191, New York, NY, USA, 2017. ACM.

[BMG17] Luciano Baresi, Danilo Filgueira Mendonça, and Martin Garriga. Empowering low-latency applications through a serverless edge computing architecture. In *Service-Oriented and Cloud Computing - 6th IFIP WG 2.14 European Conference, ESOCC 2017, Oslo, Norway, September 27-29, 2017, Proceedings*, pages 196–210, 2017.

[BMZA12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM.

[Bou18] Vanson Bourne. State Of Enterprise IT 2018. https://www.vansonbourne.com/StateOfIT2018Reports/machine-learning-and-ai, 2018. [accessed online, 17.10.2018].

[C+12] Adam Cooper et al. What is analytics? definition and essential characteristics. *CETIS Analytics Series*, 1(5):1–10, 2012.

[CCVC12] Matteo Collina, Giovanni Emanuele Corazza, and Alessandro Vanelli-Coralli. Introducing the qest broker: Scaling the iot by bridging mqtt and rest. In *Personal indoor and mobile radio communications (pimrc), 2012 ieee 23rd international symposium on*, pages 36–41. IEEE, 2012.

[Cis16] Cisco. Global Cloud Index 2016–2021. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html#_Toc503317525, 2016. [Online, accessed 12.01.2018].

[Con19a] Backstage Functions Contributors. Github ReadMe for Backstage Functions. https://github.com/globocom/functions, 2019. [Online, 14.01.2019].

[Con19b] IronFunctions Contributors. Github ReadMe for IronFunctions. https://github.com/iron-io/functions, 2019. [Online, 14.01.2019].

[Con19c] Kubeless Github Contributors. Kubeless ReadMe on Github. https://github.com/kubeless/kubeless, 2019. [Online, 14.01.2019].

[Con19d] OpenWhisk Github Contributors. OpenWhisk System Details. https://github.com/apache/incubator-openwhisk/blob/master/docs/reference.md, 2019. [Online, 14.01.2019].

[CSB17] Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya, editors. *Research Advances in Cloud Computing*. Springer Singapore, 2017.

[Das19] Dashbird. Dashbirds Features. https://dashbird.io/features/, 2019. [Online, Retrieved at 16.3.2019].

[DB18] Pavel Kravchenko David Breitgand. Lean OpenWhisk: Open Source FaaS for Edge Computing. https://medium.com/openwhisk/lean-openwhisk-open-source-faas-for-edge-computing-fb823c6bbb9b, 2018. [Online, 11.03.2019].

[DPW18] Anirban Das, Stacy Patterson, and Mike P. Wittie. Edgebench: Benchmarking edge computing platforms. *CoRR*, abs/1811.05948, 2018.

[Eiv17] A. Eivy. Be wary of the economics of "serverless" cloud computing. *IEEE Cloud Computing*, 4(2):6–12, March 2017.

[Ell18] Alex Ellis. CLI Functions with OpenFaas. https://blog.alexellis.io/cli-functions-with-openfaas/, 2018. [Online, Retrieved at 03.02.2019].

[FIMS17] Geoffrey C. Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Status of serverless computing and function-as-a-service(faas) in industry and research. *CoRR*, abs/1708.08028, 2017.

[Fis19] Fission. Fission Features. https://fission.io/features/, 2019. [Online, 14.01.2019].

[FKSH18] 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)*, pages 334–341, July 2018.

[Flo19a] Flogo. Project flogo ecosystem. https://www.flogo.io/, 2019. [accessed online, 14.01.2019 and 5.3.2019].

[Flo19b] Flogo. Project flogo github. https://github.com/tibcosoftware/flogo, 2019. [accessed online, 14.01.2019].

[Flo19c] Flogo. Project flogo inferencing. https://tibcosoftware.github.io/flogo/development/flows/tensorflow/inferencing-tf/, 2019. [accessed online, 12.10.2019].

[Fou19] Apache Foundation. Installing MXNet from MXNet: A Scalable Deep Learning Framework. http://mxnet.incubator.apache.org/versions/master/install/, 2019.

[Git19] Github. Github. https://github.com/, 2019. [Online, Retrieved at 07.01.2019].

[Gli17] Alex Glikson. Serverless Edge-to-Cloud computing: the open source way. https://medium.com/openwhisk/serverless-edge-to-cloud-computing-the-open-source-way-28ea33f60bf6, 2017. [Online, Retrieved at 15.3.2019].

[GND17] Alex Glikson, Stefan Nastic, and Schahram Dustdar. Deviceless edge computing: extending serverless computing to the edge of the network. In *Proceedings of the 10th ACM International Systems and Storage Conference*, page 28. ACM, 2017.

[Goo18a] Google. Bringing machine learning to the edge. https://cloud.google.com/blog/products/gcp/bringing-intelligence-edge-cloud-iot, 2018. [Online, Retrieved at 05.03.2019].

[Goo18b] Google. Features of Google Cloud Functions. https://cloud.google.com/functions/features/, 2018. [accessed online, 20.12.2018].

[Goo18c] Google. Google Edge Tpu. https://cloud.google.com/edge-tpu/, 2018. [accessed online, 20.10.2018].

[Goo18d] Google. Quota Limits for Google Cloud Functions. https://cloud.google.com/functions/quotas, 2018. [accessed online, 20.12.2018].

[Goo19] Google. Android Neural Networks API. https://developer.android.com/ndk/guides/neuralnetworks, 2019. [Online, 13.10.2019].

[Gre18] AWS Greengrass. *AWS Greengrass ML Inference*, 2018 (accessed July 16, 2018).

[Gro17] Aurlien Gron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 1st edition, 2017.

[GRS98] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Cure: An efficient clustering algorithm for large databases. *SIGMOD Rec.*, 27(2):73–84, June 1998.

[HFG+18] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *CoRR*, abs/1812.03651, 2018.

[HMF+17] Y. Huang, X. Ma, X. Fan, J. Liu, and W. Gong. When deep learning meets edge computing. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–2, Oct 2017.

[HPS+15] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing—a key technology towards 5g. *ETSI white paper*, 11(11):1–16, 2015.

[HTF08] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. Unsupervised learning. In *The Elements of Statistical Learning*, pages 485–585. Springer New York, dec 2008.

[(IB17] Niklas Heidloff (IBM). How to Use TensorFlow in OpenWhisk: Sample Application. https://dzone.com/articles/how-to-use-tensorflow-in-openwhisk-sample-applicat, 2017. [accessed online, 13.3.2019].

[IBM18a] IBM. IBM GPU Cloud Server Specifications. https://www.ibm.com/cloud/server-software, 2018. [accessed online, 18.10.2018].

[IBM18b] IBM. IBM GPU Cloud Servers. https://www.ibm.com/cloud/bare-metal-servers/gpu, 2018. [accessed online, 18.10.2018].

[IBM18c] IBM. System details and limits of IBM Cloud Functions. https://console.bluemix.net/docs/openwhisk/openwhisk_reference.html#openwhisk_reference, 2018. [accessed online, 20.12.2018].

[IBM18d] IBM. *Common use cases for OpenWhisk*, 2018 (accessed September, 2018).

[IBM19] IBM. IBM Cloud Functions. https://console.bluemix.net/openwhisk/, 2019. [accessed online, 15.03.2019].

[IMS17] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform. *CoRR*, abs/1710.08460, 2017.

[Int18] Intel. Intel® movidius™ neural compute stick. https://software.intel.com/en-us/neural-compute-stick, 2018. [accessed online, 20.10.2018].

[Iop18a] Iopipe. IOpipe Features. https://www.iopipe.com/features/, 2018. [accessed online, 9.1.2019].

[Iop18b] Iopipe. Iopipe supports aws lambda layers. https://read.iopipe.com/iopipe-for-aws-lambda-byol-1a184ae5a5, 2018. [accessed online, 9.1.2019].

[Jaa19] Zakaria Jaadi. A step by step explanation of Principal Component Analysis. https://towardsdatascience.com/a-step-by-step-explanation-of-principal-component-analysis-b836fb9c97e2, 2019. [accessed online, 17.06.2019].

[JWHT14] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.

[Kag17] Kaggle. 2017 The State of Data Science and Machine Learning. https://www.kaggle.com/surveys/2017, 2017. [accessed online, 23.01.2019].

[KGV17] Ashish Kumar, Saurabh Goyal, and Manik Varma. Resource-efficient machine learning in 2 KB RAM for the internet of things. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1935–1944, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[KMRR16] Jakub Konecný, H. Brendan McMahan, Daniel Ramage, and Peter Richtárik. Federated optimization: Distributed machine learning for on-device intelligence. *CoRR*, abs/1610.02527, 2016.

[Koz19] Dmitry Kozlov. OpenFaas Kubernetes TensorFlow GPU. `https://github.com/dkozlov/openfaas-tensorflow-gpu/`, 2019. [Online, 14.01.2019].

[Kra17a] Pavel Kravchenko. Apache OpenWhisk LEON. `https://github.com/kpavel/leon-openwhisk-local`, 2017. [Online, Retrieved at 15.3.2019].

[Kra17b] Pavel Kravchenko. Apache OpenWhisk light. `https://github.com/kpavel/openwhisk-light`, 2017. [Online, Retrieved at 15.3.2019].

[KZA99] Sunil Kalidindi, Matthew J. Zekauskas, and Dr. Guy T. Almes. A Round-trip Delay Metric for IPPM. RFC 2681, September 1999.

[LAP+14] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, 2014. USENIX Association.

[Lis16] Brian Liston. Analyzing Genomics Data at Scale using R, AWS Lambda, and Amazon API Gateway. `https://aws.amazon.com/de/blogs/compute/analyzing-genomics-data-at-scale-using-r-aws-lambda-and-amazon-api-gateway/`, 2016. [Online, Retrieved at 17.11.2018].

[Lit19a] TensorFlow Lite. TensorFlow Lite Guide. `https://www.tensorflow.org/lite/guide`, 2019. [Online, 12.10.2019].

[Lit19b] TensorFlow Lite. TensorFlow Lite Website. `https://www.tensorflow.org/lite`, 2019. [Online, 12.10.2019].

[LO19] Ling Liu and M. Tamer Özsu, editors. *Encyclopedia of Database Systems, Outlier Detection*. Springer New York, 2019.

[Lü17] Marco Lüthy. How to beat the AWS Lambda deployment limits. `https://hackernoon.com/exploring-the-aws-lambda-deployment-limits-9a8384b0bec3`, 2017. [accessed online, 13.3.2019].

[MAFSG17] Mehdi Mohammadi, Ala Al-Fuqaha, Sameh Sorour, and Mohsen Guizani. Deep learning for iot big data and streaming analytics: A survey, 2017.

[Mal18] Donna Malayeri. Github Ticket: Is there a limit to the deployment size? https://github.com/Azure/Azure-Functions/issues/364, 2018. [accessed online, 20.12.2018].

[Mar09] Stephen Marsland. *Machine Learning: An Algorithmic Perspective*. Chapman & Hall/CRC, 1st edition, 2009.

[MB17] G. McGrath and P. R. Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410, June 2017.

[MBR15] Roberto Minerva, Abyi Biru, and Domenico Rotondi. Towards a definition of the internet of things (iot). *IEEE Internet Initiative*, 1:1–86, 2015.

[McK18] Micah McKittrick. Can you run tensorflow-gpu with Azure Functions? https://social.msdn.microsoft.com/Forums/en-US/48bca137-4e9b-41a8-8198-d7f2489a3feb/can-you-run-tensorflowgpu-with-azure-functions?forum=AzureFunctions, 2018. [accessed online, 20.12.2018].

[MG11] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), Gaithersburg, MD, September 2011.

[Mic17a] Microsoft. Azure Functions on IoT Edge. https://blogs.msdn.microsoft.com/appserviceteam/2017/11/15/azure-functions-on-iot-edge/, 2017. [Online, 7.12.2018].

[Mic17b] Microsoft. Azure IoT Edge. https://azure.microsoft.com/en-us/services/iot-edge/, 2017. [Online, 11.03.2019].

[Mic17c] Microsoft. Azure IoT Edge Platform support. https://docs.microsoft.com/en-us/azure/iot-edge/support, 2017. [Online, 7.12.2018].

[Mic17d] Microsoft. Azure Machine Learning. https://docs.microsoft.com/en-us/azure/machine-learning/service/how-to-deploy-and-where, 2017. [Online, 18.03.2019].

[Mic18a] Microsoft. Azure Functions Python developer guide. https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference-python, 2018. [Online, Retrieved at 15.3.2019].

[Mic18b] Microsoft. Azure Functions scale and hosting. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale, 2018. [accessed online, 20.12.2018].

[Mic18c] Microsoft. Azure IoT Edge supported systems. https://docs.microsoft.com/en-us/azure/iot-edge/support, 2018. [Online, Retrieved at 7.12.2018].

[Mic18d] Microsoft. Supported Languages of Azure Functions. https://docs.microsoft.com/en-us/azure/azure-functions/functions-versions, 2018. [Online, 7.12.2018].

[Mic18e] Microsoft. Understand and use device twins in IoT Hub. https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-device-twins, 2018. [Online, Retrieved at 19.03.2019].

[Mic18f] Microsoft. Understand and use module twins in IoT Hub. https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-module-twins, 2018. [Online, Retrieved at 19.03.2019].

[Mic18g] Microsoft. What is Azure Edge. https://docs.microsoft.com/en-us/azure/iot-edge/about-iot-edge, 2018. [Online, Retrieved at 18.03.2019].

[Mic19] Microsoft. An end-to-end solution using Azure Machine Learning and IoT Edge. https://docs.microsoft.com/en-us/azure/iot-edge/tutorial-machine-learning-edge-01-intro, 2019. [Online, 13.10.2019].

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Education, 1997.

[MJS+17] A. Morshed, P. P. Jayaraman, T. Sellis, D. Georgakopoulos, M. Villari, and R. Ranjan. Deep osmosis: Holistic distributed deep learning in osmotic computing. *IEEE Cloud Computing*, 4(6):22–32, November 2017.

[MKWS07] Ilias Maglogiannis, Kostas Karpouzis, Manolis Wallace, and John Soldatos, editors. *Emerging Artificial Intelligence Applications in Computer Engineering - Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, volume 160 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2007.

[MLB+11] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Juheng Zhang, and Anand Ghalsasi. Cloud computing — the business perspective. *Decision Support Systems*, 51(1):176–189, apr 2011.

[MMRyA16] H. Brendan McMahan, Eider Moore, Daniel Ramage, and Blaise Agüera y Arcas. Communication-efficient learning of deep networksfrom decentralized data. *CoRR*, abs/1602.05629, 2016.

[MS04] Pinhas Mogilevsky and Ido Sarig. System and methods for monitoring application server performance, September 14 2004. US Patent 6,792,460.

[MW19] Chad Arimura Michael Williams, David Delabassee. Introduction to the Fn Project. https://github.com/fnproject/docs/blob/master/fn/general/introduction.md, 2019. [Online, 14.01.2019].

[ND18] Stefan Nastic and Schahram Dustdar. Towards deviceless edge computing: Challenges, design aspects, and models for serverless paradigm at the edge. In *The Essence of Software Engineering*, pages 121–136. Springer International Publishing, 2018.

[New18] Newswire. Rockchip Released Its First AI Processor RK3399Pro. https://www.prnewswire.com/news-releases/rockchip-released-its-first-ai-processor-rk3399pro----npu-performance-up-to-24tops-300578633.html, 2018. [accessed online, 20.10.2018].

[NRS+17] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21(4):64–71, 2017.

[NTD16] S. Nastic, H. Truong, and S. Dustdar. A middleware infrastructure for utility-based provisioning of iot cloud systems. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 28–40, Oct 2016.

[oO19] Github Contributors of OpenFaas. OpenFaas Github Language Templates. https://github.com/openfaas/templates, 2019. [Online, 14.01.2019].

[Ost19] Krzysztof Ostrowski. Github Issue: Installation Federated Learning on raspberry pi . https://github.com/tensorflow/federated/issues/252, 2019. [Online, 12.10.2019].

[Qar19] Halim Qarroum. A system monitoring agent for AWS Greengrass. https://github.com/HQarroum/green-sys, 2019. [accessed online, 10.5.2019].

[RC17] Roberts and Chapin. *What is Serverless*. O'Reilly Media, Inc., 2017.

[Res19] Microsoft Research. Machine learning on the edge. https://www.microsoft.com/en-us/research/project/machine-learning-edge/, 2019. [Online, Retrieved at 18.3.2019].

[RGC15] M. Ribeiro, K. Grolinger, and M. A. M. Capretz. Mlaas: Machine learning as a service. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 896–902, Dec 2015.

[RMN09] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 873–880, New York, NY, USA, 2009. ACM.

[Rob18] Mike Roberts. Serverless Architectures. `https://martinfowler.com/articles/serverless.html`, 2018. [Online, Retrieved at 24.01.2019].

[Sat17] M. Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, Jan 2017.

[SBS05] D. Steinkraus, I. Buck, and P. Y. Simard. Using gpus for machine learning algorithms. In *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, pages 1115–1120 Vol. 2, Aug 2005.

[SCZ+16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[SD16] Weisong Shi and Schahram Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, may 2016.

[Ser17] Amazon Web Services. Greengrass is now generally available. `https://aws.amazon.com/de/about-aws/whats-new/2017/06/aws-greengrass-is-now-generally-available/`, 2017. [Online].

[SL17a] J. Schneible and A. Lu. Anomaly detection on the edge. In *MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM)*, pages 678–682, Oct 2017.

[SL17b] Scikit-Learn. Scikit-Learn Website - Will You Add Gpu Support. `https://scikit-learn.org/stable/faq.html#will-you-add-gpu-support`, 2017. [Online, 18.10.2019].

[THLL09] Chih-Fong Tsai, Yu-Feng Hsu, Chia-Ying Lin, and Wei-Yang Lin. Intrusion detection by machine learning: A review. *Expert Systems with Applications*, 36(10):11994–12000, December 2009.

[Thu19a] Thundra. Thundra Features. `https://www.thundra.io/features`, 2019. [Online, 11.03.2019].

[Thu19b] Thundra. Thundra Metrics. `https://docs.thundra.io/docs/metric`, 2019. [Online, 11.03.2019].

[TRA+19] Philipp Tschandl, Cliff Rosendahl, Bengu Nisa Akay, Giuseppe Argenziano, Andreas Blum, Ralph P. Braun, Horacio Cabo, Jean-Yves Gourhant, Jürgen

Kreusch, Aimilios Lallas, Jan Lapins, Ashfaq Marghoob, Scott Menzies, Nina Maria Neuber, John Paoli, Harold S. Rabinovitz, Christoph Rinner, Alon Scope, H. Peter Soyer, Christoph Sinz, Luc Thomas, Iris Zalaudek, and Harald Kittler. Expert-level diagnosis of nonpigmented skin cancer by combined convolutional neural networks. *JAMA Dermatology*, 155(1):58, jan 2019.

[Tur02] Peter D. Turney. Thumbs up or thumbs down?: Semantic orientation applied to unsupervised classification of reviews. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 417–424, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[VPC16a] L. Valerio, A. Passarella, and M. Conti. Accuracy vs. traffic trade-off of learning iot data patterns at the edge with hypothesis transfer learning. In *2016 IEEE 2nd International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*, pages 1–6, Sep. 2016.

[VPC16b] L. Valerio, A. Passarella, and M. Conti. Hypothesis transfer learning for efficient data computing in smart cities environments. In *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–8, May 2016.

[WD92] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[Web19a] Kubeless Website. Kubeless Get Started. `https://kubeless.io/`, 2019. [Online, 14.01.2019].

[Web19b] OpenFaas Website. OpenFaas Documentation. `https://docs.openfaas.com/`, 2019. [Online, 14.01.2019].

[Web19c] OpenWhisk Website. Open Source Serverless Cloud Platform. `https://openwhisk.apache.org/`, 2019. [Online, 14.01.2019].

[WHW+18] Xiaofei Wang, Yiwen Han, Chenyang Wang, Qiyang Zhao, Xu Chen, and Min Chen. In-edge AI: intelligentizing mobile edge computing, caching and communication by federated learning. *CoRR*, abs/1809.07857, 2018.

[WLZ+18] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, 2018. USENIX Association.

[Wä16] Kai Wähner. Introduction to the Flogo Architecture. `https://www.slideshare.net/KaiWaehner/open-source-iot-`

project-flogo-introduction-overview-and-architecture, 2016. [accessed online, 5.3.2019].

[YCCI16] Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, MOTA '16, pages 5:1–5:4, New York, NY, USA, 2016. ACM.