A Fuzzy Controller for Self-adaptive Lightweight Edge Container Orchestration

Fabian Gand, Ilenia Fronza, Nabil El Ioini, Hamid R. Barzegar, Shelernaz Azimi and Claus Pahl
Free University of Bozen-Bolzano, Bolzano, Italy

Keywords: Edge Cloud, Container, Cluster, Self-adaptive, Performance Engineering, Auto-scaling, Fuzzy Logic.

Abstract: Edge clusters consisting of small and affordable single-board devices are used in a range of different applications such as microcontrollers regulating an industrial process or controllers monitoring and managing traffic roadside. We call this wider context of computational infrastructure between the sensor and Internet-of-Things world and centralised cloud data centres the edge or edge computing. Despite the growing hardware capabilities of edge devices, resources are often still limited and need to be used intelligently. This can achieved by providing a self-adaptive scaling component in these clusters that is capable of scaling individual parts of the application running in the cluster. We propose an auto-scalable container-based cluster architecture for lightweight edge devices. A serverless architecture is at the core of the management solution. Our auto-scaler as the key component of this architecture is based on fuzzy logic in order to address challenges arising from an uncertain environment. In this context, it is crucial to evaluate the capabilities and limitations of the application in a real-world context. Our results show that the proposed platform architecture, the implemented application and the scaling functionality meet the set requirements and offer a basis for lightweight edge computing.

1 INTRODUCTION

Edge Computing is defined as a concept where most processing tasks are computed directly on hardware nodes at the edge of a network and are not send to central, but remote processing hubs. The management of these systems often covers different concepts for scaling parts of the system dynamically in order to deal with changing requirements in a constrained environment. This is achieved by various means: Proposed solutions range from simple algorithms that define scale values based on set thresholds (Xi et al., 2004) to systems leveraging the power of neural networks (Lama and Zhou, 2010). Based on past research, a suitable auto-scaling algorithm needs to be suggested, implemented and evaluated for our edge context.

This work aims at evaluating whether a serverless systems managed by an intelligent auto-scaling functionality offers a potential basis for lightweight edge computing. Serverless aims at delegating the deploying, scaling and maintaining of software to the cloud/edge provider, simply requiring only to pass applications to a serverless platform. Even though a recent concept, serverless technology is already in use in a variety of application areas since it became first relevant around fivecyears ago (Baldini et al., 2017).

Our edge platform is implemented on a cluster of eight single-board devices. Small clusters have previously been evaluated in basic, i.e., static forms. However, they has not yet been used to run and evaluate a complete system based on real-life requirements and constraints with dynamic scalability included. Since small single-board devices, like the Raspberry Pi, are widely used, we will evaluate under which conditions a cluster of these devices is able to support a distributed system requiring low latency that is tightly constrained by a fixed set of performance requirements. Our system is based on MQTT for inter-cluster communication, openFaas and Docker Swarm for the implementation of the serverless concept, using Prometheus for monitoring purposes and utilizing fuzzy logic for the central auto-scaling component. The implemented auto-scaling algorithm will be evaluated experimentally. We report on the performance of the system for different scenarios.

In the remainder, in Section 2, relevant concepts, tools and technologies are introduced. Then, work on distributed edge systems and auto-scaling is discussed. Section 4 introduces first a high-level architecture, before detailing the auto-scaling component in Section 5. Finally, we evaluate the platform and conclude with directions for future research.

Gand, F., Fronza, I., El Ioini, N., Barzegar, H., Azimi, S. and Pahl, C.
A Fuzzy Controller for Self-adaptive Lightweight Edge Container Orchestration.
DOI: 10.5220/0009379600790090
In *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*, pages 79-90
ISBN: 978-989-758-424-4

2 TECHNOLOGIES & ARCHITECTURE

We introduce important concepts, tools and technologies that we combine to define the core architecture.

2.1 Architecture Concepts

The presented platform is based on a range of concepts for architecture, deployment and self-adaptivity.

Cloud computing is based on centralised data centers that are able to process large amounts of data in the "Cloud" (Kiss et al., 2018). This, however, leaves the potential local processing power of the "Edge" network unused, causing often significantly increased latency. Edge Computing leverages the processing power of local nodes at the edge of the network. These nodes are an intermediate layer (Kiss et al., 2018) that process data within the local network that would otherwise by handled by a remote cloud.

Microservices are a recent architectural paradigm. Traditional architectures usually deliver an application as a monolith, i.e., an application is bundled into one executable that is then deployed. When migrating to a microservice architecture, the monolith is split into different parts (the microservices) that run in independent processes with their own deployment artifacts (Jamshidi et al., 2018).

The serverless concepts allows developers to focus only on the application without having to consider the deployment servers, which is handled by the cloud provider. This allows for features such as fault-tolerance and auto-scaling to be managed (Baldini et al., 2017). Usually, serverless computing is linked to a concept called Functions-as-a-Service (FaaS), which means that small chunks of functionality are deployed in the cloud and scaled dynamically by the cloud provider (Kritikos and Skrzypek, 2018). These functions are usually smaller than microservices, are short-lived and have clear input and output parameters. If the component to be deployed is more complex than a simple function and is supposed to stay active for a longer period of time, a stateless microservice is another option (Ellis, 2018).

2.2 Self-adaptiveness and Fuzzyness

A self-adaptive system dynamically adapts its behavior to either preserve or enhance required quality attributes in the presence of uncertain operating conditions. The development of microservice applications as self-adaptive systems is still a challenge (Mendonca et al., 2019). In practice, e.g., the Kubernetes container orchestration platform facilitates to deploy and manage microservice applications, but natively only supports basic autoscaling by automatically change the number of instances of a service.

Computers traditionally excel at tasks that contain simple mathematical calculations. Human reasoning, however, is usually more complex. Natural language is rarely precise in a way that it quantifies something as one thing or the other: words can have uncertain, ambiguous meanings (Hong and Lee, 1996), human reasoning is fuzzy. Uncertainty also arises in edge environments through incomplete or potentially incorrect or conflicting observations. Fuzzy logic is trying to represent this by mapping inputs (e.g., observations) to outputs (e.g., analyses or reactions) based on gradually changing functions and a set of rules rather than fixed thresholds. A *membership function* is a function that represents a fuzzy set and decides to which degree an item belongs to a certain set. *Fuzzification* refers to input values that are mapped to membership functions to derive the degree of membership in that set (Lin and Lee, 1991). This is a stark contrast to a binary approach where the element can either be part of a set or not. *Fuzzy rules* define how after fuzzification the values are matched against if-else rules. *Defuzzyfication* is the final step, where a numerical output value is generated.

A sample goal might be to calculate the money that should to be saved each month based on a flexible salary and the expected expenses. The amount of money that is to be saved in a long-term savings account will be returned by the fuzzy system.

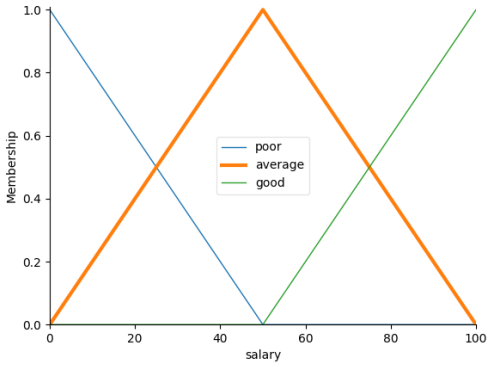
- *Membership functions* – The three variables salary, expected expenses and the money that is suggested to be saved are visualized in Figures 1a, 1b and 2. Each variable consists of three membership functions: low, high and medium – representing, for example, the degree to which a salary of 50 can be considered a low, high or average salary.
- *Rules* – The rules are set as follows:

```
IF salary LOW OR expenses HIGH THEN savings LOW
IF salary AVERAGE THEN savings AVERAGE
IF salary HIGH OR expenses LOW THEN savings HIGH
```

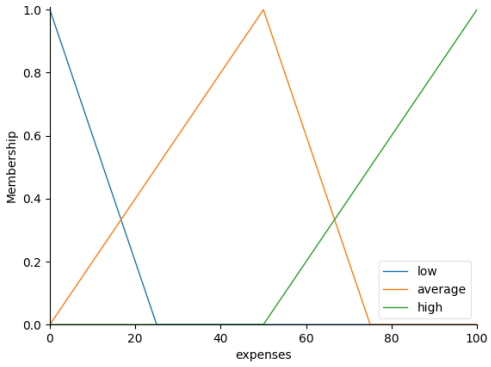
- *System Inputs* – The savings are calculated based on a salary of 82 and expected expenses of 51.4.
- *Defuzzyfication* – After defuzzyfication, we get a savings recommendation of 15.03.

2.3 Selected Tools and Technologies

The concrete infrastructure and software technologies used in the implementation of the platform shall be introduced now. The *Raspberry Pi* is a single-board computer based on an ARM-processor. *Docker* is a



(a) Membership function of the flexible salary.



(b) Membership functions of the expected expenses.

Figure 1: Membership functions of the two input values.

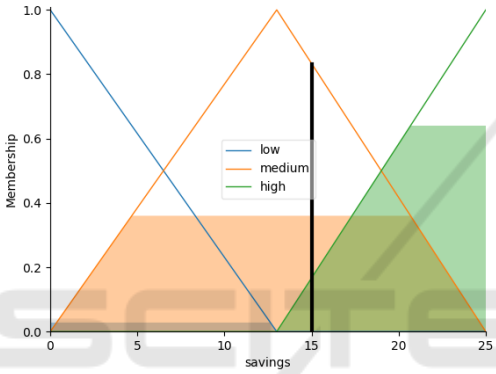


Figure 2: Savings membership functions with final value.

containerization software. Containerization is a virtualization technology that, instead of virtualizing hardware, separates processes from each other. It has been successfully used on Raspberry Pis (Scolati et al., 2019). *Docker swarm* is the cluster management tool that is integrated into the Docker Engine. It is based on services. Instead of running the services and their corresponding containers on one host, they can be deployed on a cluster of nodes that is managed like a single, docker-based system. By setting the desired number of replicas of a service, basic scaling can also be handled. *MQTT* is a network protocol suitable for the Internet of Things. *Prometheus* is a monitoring tool that can be used to gather and process application metrics. Contrary to other monitoring tools, Prometheus "scrapes" the metrics from a predetermined interface in a given interval. OpenFaas is a Functions-as-a-Service framework that can be deployed on top of a Docker swarm or a Kubernetes cluster. By default openFaas contains simple autoscaling that leverages the default metrics aggregated by Prometheus and scales based on predefined thresholds (openFaaS, 2019). Stateless microservices can be built by using the *Dockerfile* template.

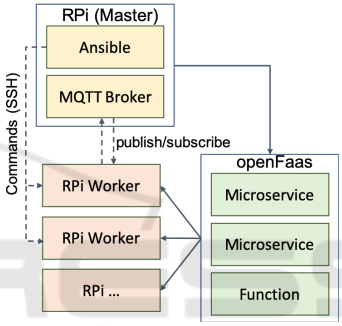


Figure 3: Hardware and networking components.

2.4 Lightweight Edge Platform Design

Combining the above technologies suggests splitting the application into microservices and containerizing it, allowing the hardware to be reallocated dynamically. This also enables scaling the different parts of the application. OpenFaas allows building and deploying services in the form of functions across the cluster. Some of its features, such as built-in Prometheus or the gateway, can be extended upon to make openFaas the central building block of the application (Gand et al., 2020). An overview of the different technologies in the context of the given platform is provided in Figure 3. Even though openFaas scaling is limited, it can be used as a foundation for our self-adaption. We implement a more fine-grained scaling algorithm using the built-in monitoring options as well as fuzzy logic as the reasoning foundation.

3 RELATED WORK

A system similar to ours in architectural terms as a Raspberry Pi-based implementation of an application system has been introduced in (Steffenel et al., 2019). The authors introduce a containerized cluster based on single-board devices that is tailored towards applications that process and compute large amounts of data. They deploy this application, a weather forecast model, to the Raspberry Pi cluster and evaluate its performance. They note that the performance of the RPI cluster is within limits acceptable and could be a suitable option for comparable use cases, although networking performance of the Raspberry Pis has been identified as a bottleneck. We will address performance degradations here through auto-scaling.

There are a range of scalabity approaches. (Jamshidi et al., 2014) categorize them into reactive, proactive and hybrid approaches. Reactive approaches react to changes in the environment and take action accordingly. Proactive approaches predict the need for resources in advance. Hybrid systems combine both approaches, making use of previously collected data and resource provisioning at run time.

AI approaches in this context can be used to improve the response of the network to certain environmental factors such as network traffic or threats to the system (Li et al., 2017). The authors propose a general architecture of smart 5G system that includes an independent AI controller that communicates with the components of the application. Examples of implementations of algorithms for auto-scaling and auto-configuration exist. (Saboori et al., 2008) propose tuning configuration parameters using an evolutionary algorithm called Covariance-Matrix-Adaption. Using this approach, potential candidates for solving a problem are continuously evolved, evaluated and selected, guiding the subsequent generations towards a desired outcome. Another paper introduces a Smart Hill Climbing Algorithm for finding the optimal configuration for a Web application (Xi et al., 2004). The proposed solution is based on two phases: In the global search phase, they broadly scan the search space for a potential starting point of the local search phase. Another interesting approach aims at optimizing the configuration of Hadoop, a framework for distributed programming tasks (Kambatla et al., 2009), an offline, proactive tuning algorithm that does not reallocate resources at run-time, but tries to find the best configuration in advance.

There are some proposals using fuzzy logic for auto-configuration. (Jamshidi et al., 2014) introduce an elasticity controller that uses fuzzy logic to automatically reallocate resources in a cloud-based system. Time-series forecasting is used to obtain the estimated workload at a given time in the future. Based on this information, the elasticity controller calculates and returns the number of virtual machines that needs to be added to or removed from the system. The allocation of VMs is consequently carried out by the cloud platform. There, the rule base and membership functions are based on the experience of experts.

Reinforcement Learning in the form of Fuzzy Q-learning has been evaluated in this context (Ipek et al., 2008), (Jamshidi et al., 2015). The aim of these systems is to find the optimal configuration provisioning by interacting with the system. The controller, making use of the Q-learning algorithm, selects the scaling actions that yield the highest long-term reward. It will also keep updating the mapping of reward values to state-action pairs (Ipek et al., 2008). Adaptive neuro-fuzzy inference systems (ANFIS), combining neural networks and fuzzy logic, are discussed in (Jang, 1993). Fuzzy logic and neural networks are shown to complement each other in a hybrid system.

The work presented in this paper focuses on applying a practical approach for implementing a lightweight auto-scaling controller based on fuzzy logic. So far, fuzzy auto-scaling for lightweight edge architectures has not been investigated.

4 ARCHITECTURE OVERVIEW

We overview the architecture before covering low-level implementation details of the platform and its auto-scaling component. The proposed binding blocks of the application such as the a serverless and microservices-based architecture can be reused for different applications in different contexts. The scaling component is also usable in different applications by simply updating a few constants.

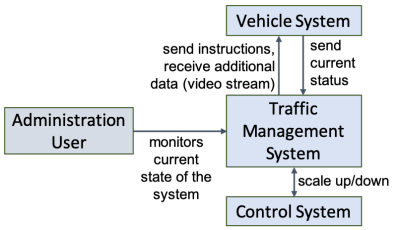


Figure 4: Interaction between different systems.

The architecture consists of three layers. The platform layer represents the hardware architecture of the cluster. The system layer comprises the central management components. On top of these, the controller layer scales the components of the platform. Figure 4 shows the interaction between system layer, controller

layer and additional components. We use a Traffic Management (TM) System as a sample application. We assume a constant exchange of messages between the traffic management and vehicle components that in our prototype implementation contains simulations of vehicles. The control system is used to scale the TM System based on the current situation.

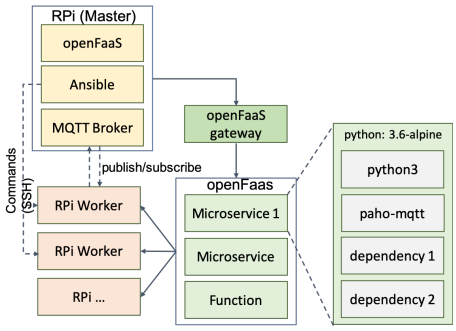


Figure 5: Platform Layer

The high-level platform layer is shown in Figure 5. The application is deployed on a cluster managed by Docker Swarm. The cluster includes one master node and an arbitrary number of worker nodes. Ansible is used to execute commands on all nodes without having to connect to each node individually. All nodes are able to connect to the MQTT broker that is running on the master device after startup. Using Docker swarm and openFaas, the RPIs can be connected so that they can be seen as one system. If a service is supposed to be deployed, openFaas will distribute it among the available nodes. There is no need to specify a specific node as this abstraction layer is hidden behind the openFaas framework. The services and functions are built and deployed using the openFaas command line interface. OpenFaas is also utilized to scale the services independently. Communication between the services is achieved by relying on the openFaas gateway as well as on the MQTT broker. The cluster is comprised of eight Raspberry Pi 2 Model B connected to a mobile switch via 10/100 Mbit/s Ethernet that is powering the RPIs via PoE (Power over Ethernet).

Monitoring is done through the openFaas Prometheus instance. This instance is used to store metrics and query them when needed. Before startup, Prometheus needs to be informed about the endpoints that metrics should be collected from.

5 AUTOSCALING CONTROLLER

The autoscaling controller is the central component in the architecture to manage performance. One requirement for it was that it had to be lightweight. Using too many system resources such as storage space or CPU usage was to be avoided since the algorithm is meant to be deployed on the RPI cluster itself with most of the clusters resources being reserved for actual application components.

Fuzzy logic allows for a good compromise between a powerful decision-making process and a limited resource consumption. However, the initial fuzzy knowledge base is difficult to obtain. The following solution combines reactive and proactive configuration methods by initially anticipating demand in the calibration and configuration (proactive) and then continuously re-adjusting them if needed (reactive). Values of previous runs of the system are used to set-up an initial fuzzy knowledge base. The reactive part of the algorithm continuously updates parts of the knowledge base at runtime.

5.1 Scaling Principles

The functionality of the algorithm can be visualized by using the MAPE-K (Kephart and Chess, 2003) controller loop for self-adaptive systems. The steps of the controller loop are as follows: *Monitor* the application and collect metrics, *Analyze* the gathered data, *Plan* actions accordingly in order to maintain objectives, and *Execute* the planned actions. The *Knowledge* component defines a shared knowledge base that is continuously updated.

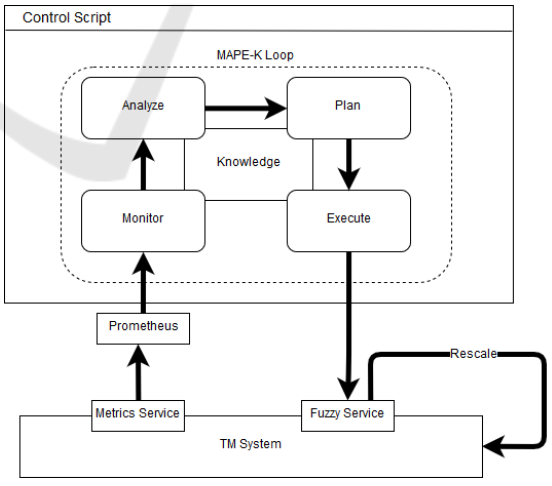


Figure 6: MAPE-K loop for fuzzy auto-scaling controller.

Scaling Configuration and Calibration: The MAPE-K loop for the given system is presented in Figure 6. The scaling algorithm, based on the four main phases of the loop, is implemented in a python script (*control.py*). The script is run independently on a single cluster node. The algorithm starts by build-

ing the fuzzy membership functions, essentially calibrating them based on existing experience. The values used for constructing the functions are part of the MAPE-K knowledge base. These values are calculated by relying on metrics of previous runs of the system. Therefore, before starting the scaling algorithm effectively, the system needs have been run at least once to detemine behaviour that can be anticipated. Based on the initial membership functions, a first global scale value is computed according to which the system is scaled. This forms the *proactive* part of the controller, which provides settings for future runs based on anticipated load and performance.

Continuous Scaling: Once the system is started, the *reactive* part of the algorithm is executed as the default that adjusts current settings to specified requirements. The script receives current performance metrics from the Prometheus API. These metrics are evaluated against the allowed threshold values stored in the knowledge component. The knowledge component is implemented within the control script. Based on these computations, a plan is devised that involves updating the fuzzy membership functions. The goal is to constantly update the membership function such that the fuzzy service provides optimal scale values for all load scenarios. Here, optimal is defined as a system that is scaled in order to meet the SLO without wasting unnecessary resources, i.e., the ultimate goal is to only scale up as much as necessary. The definitions of the membership functions are also part of the knowledge component and are continuously updated. The membership functions are passed to the fuzzy service that calculates a scale value. After scaling, the loop repeats by monitoring and analyzing the effects of the previous iteration.

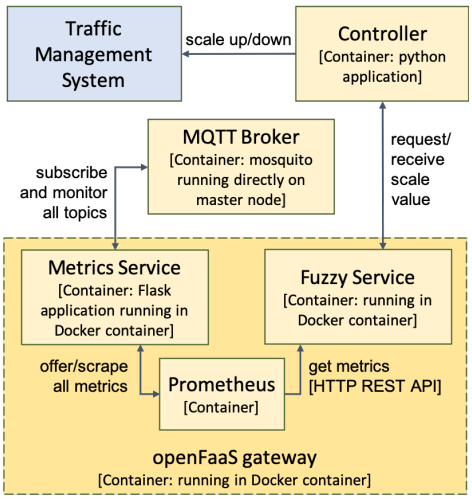


Figure 7: Details of services providing scaling functionality.

5.2 The Fuzzy Service

The *fuzzy service* is used to calculate the scale value. It receives the *membership functions as input*. Another parameter is the range of the scale values. This parameter can be defined to set the minimum and maximum scale values that are perceived as acceptable. The services returns a *global scale value as output*. The rules in the *fuzzy service* are predefined. The minimal alpine distribution, which is used for the other images, could not be used in this case since *skfuzzy*, the python fuzzy module that was used, relies on the *numpy* module that requires a more complex OS. Therefore, the image of the *fuzzy service* is based on *ubuntu:18.04*. The *calc* function is used to simulate a function that is called continuously. It is always scaled to the maximum. If there are only a few message processes by the system that the *calc* function is allowed to scale higher. When the number rises, the allowed number of replicas will decrease.

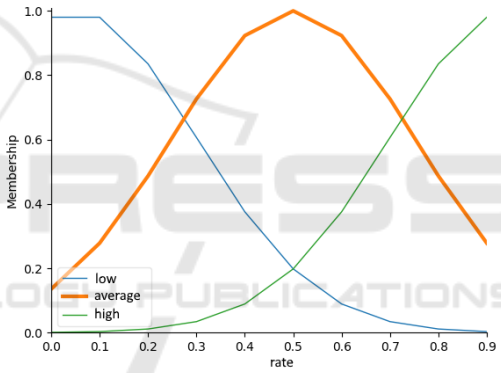


Figure 8: Initial membership functions.

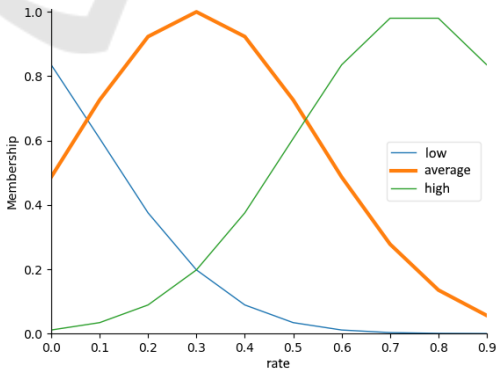


Figure 9: Re-adjusted membership functions: the system can handle less messages than expected.

5.3 Calibration and Configuration

This preparatory part of the solution relies on metrics gathered during previous runs of the system that are then used as the basis for calibrating the fuzzy membership functions to manage anticipated demand. The metric that is used to measure the current load of the system is the number of exchanged messages in a given time frame. This metric, called *messages_total*, only defines the total number of messages the system had to process since the monitoring started. What is needed, is a way to measure how many messages were processed in a given timeframe. We base this on the Prometheus *rate()* function, which returns the "per second rate of increase". Executing the following example query that is used throughout the application returns the per-second rate of increase measured over the last 20 seconds:

```
rate(messages_count_total[20s])
```

The rate r is calculated as follows:

$$r = \frac{x_i - x_{i-t}}{t} \tag{1}$$

where x_{i-t} is the number of messages for t seconds in the past with x_i as the most recent number of messages. This value is divided by t, the time frame that is considered, since r is the per-second value. Based on the rate, the Prometheus query language is used to obtain three values: the global median, the global standard deviation and the global maximum. Global in this context refers to metrics that were calculated based on previous runs of the system. These values are used to create the initial membership functions of the rate of messages. In *sk-fuzzy*, Gaussian membership functions are created by defining their mean and their standard deviation. The Gaussian membership functions are simple to create and re-adjust. All membership functions set the previously obtained global standard deviation σ_g as their own standard deviation. The mean of the *average* membership function is placed at the global median:

$$median(a)_g \tag{2}$$

with a being a set of data, in this case the rate of messages. The mean of the *low* membership function is:

$$median(a)_g - \sigma_g \tag{3}$$

with σ_g again being the global standard deviation. The mean of the *high* membership function is thus:

$$median(a)_g + \sigma_g \tag{4}$$

The initial membership functions are shown in Fig. 8.

The details of calculating the three global variables are discussed now. The following sample query returns the single median value over all rates of messages (the rates covering a time span of 20 seconds each) of the last 90 days:

```
quantile_over_time(
  0.5,
  rate(messages_count_total[20s])[90d:20s]
)
```

Prometheus does not offer a *median* function, but the built-in *quantile* function can be used to calculate the median. A quantile splits a proabablilty distribution into groups according to a given threshold. The 0.5 quantile equals the median. Generally, the median can be expressed as follows:

$$Q_A(0.5) = median(a) \tag{5}$$

The standard deviation over time is calculated by using the corresponding *PromQL* function:

```
stddev_over_time(
  rate(messages_count_total[20s])[90d:20s])
)
```

This call returns the standard deviation of the rate of messages (covering a time span of 20 seconds each). The total time span that is considered are the last 90 days. The standard deviation for the given context is:

$$\sigma_g = \sqrt{\frac{1}{N} * \sum_{i=1}^{N} *(x_i - \mu)} \tag{6}$$

with N being the rates of messages considered while x_i is a single rate of messages and μ is the mean of all rates of messages. Finally, only the global maximum of the data set remains to be calculated:

```
max_over_time(
  rate(messages_count_total[20s])[90d:20s]
)
```

5.4 Continuous Scaling

The continuous scaling uses the previously created initial membership functions and re-adjusts them continuously. This reactive part is implemented as part of an infinite loop. After the initial configuration and calibration part is concluded, the initial membership functions are passed to the fuzzy service that returns a scale value. This is used to scale services as follows:

$$s = round(D * G) \tag{7}$$

where s is the new scale value a specific service is scaled to, D is the default value the service is scaled to at startup and G is the rounded global scale value that has been calculated by the fuzzy service and is used for all dynamically scaled components. These

scalable components include the gatherer service and the decision function.

After scaling the components and waiting for a set period of time, the Message Roundtrip Time MRT is calculated. This metric for the sample Traffic Management use case indicates the average time a vehicle needs to wait for a response from the gatherer after publishing its latest status. In the analysis phase, the MRT is compared to the maximum threshold that is defined in the SLA. If the average MRT is above the SLO, the membership functions need to be re-adjusted. The initial membership function shown in Fig. 8 results in a scale value too low for the given load: the measured MRT after scaling was above the defined threshold. Therefore, the membership function need to be shifted to the left. The result of this shift can be seen in Fig. 9. If we assume the current load to be 0.5 and compare the degree of membership in both figures, we find that in the initial figure a load value of 0.5 is considered an "average" load. Looking at the re-adjusted functions, a value of 0.5 is now seen as more of a "high" load. Since the fuzzy service classifies a value of 0.5 as a "higher" load now, it also maps it to a higher scale value. Consequently, the system now scales up at a load value of 0.5 where it had previously taken no action. Similarly, we need to consider a situation in which the MRT is well below the defined threshold. In this case, the functions need to be shifted to the right. The rate at which the functions are shifted in each iteration of the loop can be controlled by (manually) updating the adjustment factor. Hence, the membership functions for each iteration are computed as follows:

- Membership function *low*: $median(a)_g - \sigma_g + A$
- Membership function *average* : $median(a)_g + A$
- Membership function *high*: $median(a)_g + \sigma_g + A$

where A here is the adjustment factor for the functions. A is computed at each iteration. A positive re-adjustment factor results in a shift to the right, a negative one shifts functions to the left. To avoid constantly moving the functions back and forth, shifting right is only allowed until a situation is encountered where the STO threshold is not met anymore. After completing the shifting process within the control script, the reactive part of the algorithm is started anew by passing the re-adjusted membership functions to the fuzzy service.

The following algorithm provides an overview of the continuous scaling functionality:

Algorithm 1: Reactive Autoscaling.

```
1: function SCALE(MEMBERSHIPFCTS)
2:     sto ← STO threshold
3:     scalevalue ←
4:      getScaleValueFromFuzzyService(membershipFcts)
5:     rescale(scalevalue)
6:     invocationtime ← get current invocation time
7:     if invocationtime < slo then
8:         shift membership functions right
9:     if invocationtime >= slo then
10:        shift membership functions left
11:    Scale(membershipFcts)
```

6 EVALUATION

The evaluation focus lies on the performance of the proposed serverless microservice solution and the auto-scaling approach. The bottlenecks are identified by measuring a number of performance metrics that will be introduced later on.

6.1 Evaluation Objectives and Metrics

The effectiveness of the autoscaling approach needs to be evaluated as the core solution component. Being effective means that the auto-scaling algorithm is able to maintain the set SLO thresholds by re-adjusting the fuzzy membership functions. This would result in a smooth scaling process where the scalable components are gradually scaled up or down, avoiding sudden leaps of the scale value. For the given application, we alslo determined the maximum load, i.e., here the maximum number of vehicles the architecture (including the network it is operated in) can support. Note, that the vehicles referred to here could be replaced by other application components.

These objectives were evaluated for two different cluster set-ups. In order to obtain a first understanding of the system and the possible range of variables, a pilot calibration evaluation was conducted on a small cluster of three RPIs. Then, the evaluation procedure was repeated for a complete cluster of eight devices.

All evaluation steps report on a number of performance metrics that indicate the effectiveness of the system or provide insight into an internal process. The Message Roundtrip Time (MRT) is the central variable of the system since it reports on the effectiveness of the application execution. Included in the MRT is the (openFaas) Function Invocation Time (FIT) that is listed separately in order to individually report on serverless performance aspects. All MRT and FIT values are considered average values aggregated over the last 20 seconds after the previous scal-

Table 1: Results for cluster of 3 RPIs. Scaling was active with variables set to values reported in Figure 2.

Vehicles	Memory	CPU	MRT	FIT
2	25.55	46.97	1.63	1.51
4	37.94	47.24	1.85	1.48
8	57.1	49.36	1.79	1.61
12	71.77	51.81	4.49	1.79
14	92.78	55.49	10.24	2.13

Table 2: Initial Calibration Scaling run for a cluster of three RPIs with a (fixed) number of 12 cars. Shift Left/Right reports on the direction the membership functions are shifted to during a given iteration. The number of scaled components is also reported. Manually set variables: Maximum Scale Value: 5, MRT Threshold: 2.0 seconds – FSV: Fuzzy Scale Value, IT: Invocation Time after scaling, Ga: Gatherer, DF: Decision Function.

Iteration	FSV	IT	Shift	# Ga	# DF
1	2.74	1.74	Right	10	5
2	2.63	2.5	Left	10	5
3	2.74	2.05	Left	10	5
4	2.95	1.9	Right	12	6

ing operation was completed. Here, the maximum scale value was unknown. In concrete scenarios, this value might have been specified beforehand.

We use different MRT thresholds. In concrete settings, the maximum response time could be given in advance and would unlikely to be the subject of change. For all set-ups and iterations that were evaluated, the hardware workload was measured by computing the average CPU and memory usage over all nodes of the cluster, combining it to a single value.

6.2 Evaluation Setup

In order to evaluate the system, a couple of configurations had to be made. Prometheus is used to gather and aggregate the metrics for the evaluation. The cloud-init configuration was adjusted so that an additional python script is executed on each node. The script uses the *psutil* python module to record CPU and memory usage of the node and publish them to a specific MQTT topic. Additional functionality was also added to the metrics service: the service receives the CPU/memory metrics of all nodes and stores them internally. When the metrics are collected, the service calculates average CPU and memory usage across all nodes and exposes them to Prometheus.

6.3 Experimental Evaluation

Pilot Calibration. An initial evaluation (calibration pilot) was conducted to obtain an initial idea of the system's capabilities and adjust the manually-tuned parameters accordingly. It was also used to evaluate whether the scaling functionality yields promising results before putting it to use in a bigger set-up. The evaluation was started with a cluster consisting of three RPIs: a master and two worker nodes. The maximum scale value was initially set to 5 in order to avoid scaling unreasonably high. Table 1 reports on the initial set of metrics for different numbers of vehicles. The scaling functionality was active when the metrics were monitored. Table 2 includes data of the scaling algorithm for an initial calibrating run.

Table 3: Scaling experiment for cluster of 8 RPIs. Manually set variables: Maximum Scale Value: 12, Number of cars: changing, MRT Threshold: 0.3 seconds – FSV: Fuzzy Scale Value, AF: Adjustment Factor.

Iteration	cars	FCV	MRT	AF
1	25	4.77	Initial	0.0
2	25	6.94	0.03	2.58
3	25	7.79	0.04	0.0
4	25	8.80	0.03	-2.58
5	50	10.11	0.178	-7.74
6	50	10.82	0.44	-10.32
7	50	10.82	0.44	-10.32
8	75	11.06	0.09	-12.9
9	75	11.24	0.033	-18.06
10	75	11.3	3.95	-20.64
11	75	11.34	0.07	-23.22

Full Cluster. A cluster of eight RPIs was used. Here, the decision-making functionality is included in the gatherer service, which is now scaled independently. Hence, there is no longer the need to call the decision function for each message. Results for the auto-scaling algorithm can be found in Table 3.

The CPU usage started at 47% and showed a linear increase up to about 56% at 14 vehicles. The hardware does not seem to be the limiting factor. The inital setup, however, shows a MRT of about 1.6 seconds for only 2 vehicles. At 12 vehicles, a Roundtrip Time of 4.5 seconds is reached and at 14 vehicles the MRT is already above 10 seconds, which is clearly a value too high for many real-world scenario. However, based on these findings, an initial evaluation run of the auto-scaling algorithm was conducted with the Roundtrip Time threshold set to 2.0 seconds. Even though this is not realistic, it allows for testing the auto-scaling functionality.

The experimental runs yielded promising results and show that the algorithm is able to adaptively rescale the system: The MRT is initially below the predefined threshold at about 1.7 seconds. In the second iteration, the MRT value is recorded above the threshold at 2.5 seconds. The system reacts to this situation by slowly re-adjusting the membership function, resulting in a higher fuzzy scale value, which consequently scales the system up until the measured

MRT dropps below the threshold again.

6.4 Discussion of Evaluation Results

Using a smaller cluster set-up, it was possible to derive starting values for all variables (calibration pilot).

The evaluation of the architecture indicated that serverless function calls should be restricted since they introduce network latency problems. The limiting factor is here the network. The used set-up was not able to process more than 75 user processes at a time. The CPU and Memory usage numbers as well as the steady but slow increase of the MRT imply that the hardware itself is able process a higher number of vehicles. Future evaluations should find different network set-ups that allow for a dependable service beyond identified limits.

The implemented scaling algorithm works as intended and is able to scale the system in a balanced manner. After finding a SLO, that is usually given by the specifications of the system, the only values that need to be set manually are the maximum scale value and the adjustment factor.

In summary, the full version of the system yields satisfying results in terms of hardware consumption and performance (MRT), while the presented scaling algorithm gradually scales the system as intended.

7 CONCLUSIONS

This work introduced a containerized platform architecture on a cluster of single-board devices. The applied approach results in a reconfigurable, scalable and dependable system that provides built-in solutions for common problems such as service discovery and inter-service communication. The implementation is a proof-of-concept with the constraints of the environment playing a crucial factor in the implementation. We aimed at experimentally evaluating a self-adaptive autoscaler based on the openFaas framework and a microservices-based architecture. The evaluation of the system was conducted to determine the performance of the proposed set-up. The implemented auto-scaling algorithm was specifically evaluated in order to assess dependable resource management for lightweight edge device infrastructures. By using openFaas beyond its documented boundaries, it was possible to use the framework for inter-service communication as well as for monitoring.

The auto-scaling algorithm, by using fuzzy logic, is able to gradually scale the system. Unlike previous examples of fuzzy auto-scalers that were deployed to a large cloud infrastructure, the fuzzy scaling functionality here was constrained in its processing demands since it was deployed alongside the main system components on the limited hardware cluster. Consequently, the algorithm was focused on using data and technologies that were already available within the cluster. There still remain parameters that need to be tuned manually to achieve the desired outcomes. This leaves a certain degree of uncertainty when applying the algorithm on a black-box system. The evaluation also showed that the given set-up is only able to process up to 75 vehicles simultaneously in the used traffic management application, but indicating network aspects as the reason (while our focus was on compute/storage resources).

Future work shall focus on improving the application management components to drive them towards a more realistic behavior. Additionally, security concerns have not been covered in detail and need more attention in the future. However, this abstract application scenarios, with cars just being simulated data providers, allowed us to generalise the results beyond the concrete application case.

This also applies for fuzzy systems based on neural networks. Related work on Adaptive Neuro-Fuzzy Inference Systems (ANFIS) was discussed earlier. Working examples of ANFIS are relatively rare. One example uses python and the machine-learning framework tensorflow to combine fuzzy logic and a neural network (Cuervo, 2019). Relying on the ubuntu distribution used for the fuzzy service, it is possible to create an ANFIS service for our setting. Previously, compiling and running the tensorflow framework on hardware such as the Raspberry Pi was a challenging task. With release 1.9 of the framework, tensorflow officially started supporting the Raspberry Pi (Warden, 2019). Pre-built packages for the platform can now be directly installed. This adds new possibilities for creating a lightweight ANFIS service for the given system. ANFISs rely on training data. This introduces the challenge of finding sample data that maps the rate of messages and a scale value to an indicator that classifies the performance as acceptable/not acceptable: *(rate of messages, scale value)* $\rightarrow$ *acceptable/not acceptable performance* This dataset could be aggregated manually or in an automated manner.

Another direction is to address more complex architectures with multiple clusters. We propose Particle Swarm Optimization (PSO) (Azimi et al., 2020) here, which is a bio-inspired optimization methods suitable to coordinate between autonomous entities such as clusters in our case. PSO distinguishes personal (here local cluster) best fitness and global (here cross-cluster) best fitness in the allocation of load to

clusters and their nodes. This shall be combined with the fuzzy scaling at cluster level as proposed here.

In terms of use cases, we looked at traffic management and coordinated cars, where traffic and car movement is captured and processed, maybe supplemented by infotainment information with image and video data. Another application domain is mobile learning that equally includes heavy use of multimedia content being delivered to mobile learners and their devices (Murray et al., 2003; Pahl et al., 2004; Kenny et al., 2003). These systems also rely on close interaction with semantic processing of interactions in order to support cognitive learnign processes. Some of these can be provided at edge layer to enable satisfactory user experience.

ACKNOWLEDGEMENTS

This work has received funding from the EU's Horizon 2020 research and innovation programme under grant agreement 825012 - Project 5G-CARMEN.

REFERENCES

Ardagna, C. A., Asal, R., Damiani, E., Dimitrakos, T., El Ioini, N. and Pahl, C. (2018). Certification-based cloud adaptation. *IEEE Transactions on Services Computing*.

Azimi, S., Pahl, C., and Shirvani, M. H. (2020). Particle swarm optimization for managing performance in multi-cluster IoT edge architectures. In *International Conference on Cloud Computing and Services Science CLOSER*.

Baldini, I., Castro, P. C., Chang, K. S., Cheng, P., Fink, S. J., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R. M., Slominski, A., and Suter, P. (2017). Serverless computing: Current trends and open problems. *CoRR*, abs/1706.03178.

Cuervo, T. (2019). TensorANFIS. https://github.com/tiagoCuervo/TensorANFIS. Accessed: 2019-11-04.

Ellis, A. (2018). Introducing stateless microservices for openfaas. https://www.openfaas.com/blog/stateless-microservices/.

Fang, D., Liu, X., Romdhani, I., Jamshidi, P. and Pahl, C. (2016). An agility-oriented and fuzziness-embedded semantic model for collaborative cloud service search, retrieval and recommendation. In *Future Generation Computer Systems*, 56, 11-26.

Fowley, F., Pahl, C., Jamshidi, P., Fang, D. and Liu, X. (2018). A classification and comparison framework for cloud service brokerage architectures. IEEE Transactions on Cloud Computing 6 (2), 358-371.

Gand, F., Fronza, I., El Ioini, N., Barzegar, H. R., and Pahl, C. (2020). Serverless container cluster management for lightweight edge clouds. In *10th Intl Conf on Cloud Computing and Services Science CLOSER*.

Gand, F., Fronza, I., El Ioini, N., Barzegar, H. R. and Pahl, C. (2020). A Lightweight Virtualisation Platform for Cooperative, Connected and Automated Mobility. In *6th International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS)*.

Hong, T.-P. and Lee, C.-Y. (1996). Induction of fuzzy rules and membership functions from training examples. *Fuzzy Sets Syst.*, 84(1):33–47.

El Ioini, N. and Pahl, C. (2018). A review of distributed ledger technologies. OTM Confederated International Conferences.

El Ioini, N. and Pahl, C. (2018). Trustworthy Orchestration of Container Based Edge Computing Using Permissioned Blockchain. Intl Conf on Internet of Things: Systems, Management and Security (IoTSMS).

El Ioini, N., Pahl, C. and Helmer, S. (2018). A decision framework for blockchain platforms for IoT and edge computing. IoTBDS'18.

Ipek, E., Mutlu, O., Martinez, J. F., and Caruana, R. (2008). Self-optimizing memory controllers: A reinforcement learning approach. In *2008 International Symposium on Computer Architecture*, pages 39–50.

Jamshidi, P., Pahl, C., Chinenyeze, S. and Liu, X. (2015). Cloud Migration Patterns: A Multi-cloud Service Architecture Perspective. In *Service-Oriented Computing - ICSOC 2014 Workshops*. 6–19.

Jamshidi, P., Sharifloo, A., Pahl, C., Arabnejad, H., Metzger, A. and Estrada, G. (2016). Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. *Intl Conf Quality of Software Architectures*, 70–79.

Jamshidi, P., Pahl, C. and Mendonca, N. C. (2016). Managing uncertainty in autonomic cloud elasticity controllers. IEEE Cloud Computing, 50-60.

Jamshidi, P., Pahl, C. and Mendonca, N. C. (2017). Pattern-based multi-cloud architecture migration. Software: Practice and Experience 47 (9), 1159-1184.

Jamshidi, P., Ahmad, A., and Pahl, C. (2014). Autonomic resource provisioning for cloud-based software. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014.

Jamshidi, P., Pahl, C., Mendonca, N. C., Lewis, J., and Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35.

Jamshidi, P., Sharifloo, A. M., Pahl, C., Metzger, A., and Estrada, G. (2015). Self-learning cloud controllers: Fuzzy q-learning for knowledge evolution. In *ICAC'2015*, pages 208–211.

Jang, J. S. R. (1993). Anfis: adaptive-network-based fuzzy inference system. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(3):665–685.

Javed, M., Abgaz, Y. M. and Pahl, C. (2013). Ontology change management and identification of change patterns. Journal on Data Semantics 2 (2-3), 119-143.

Kambatla, K., Pathak, A., and Pucha, H. (2009). Towards optimizing hadoop provisioning in the cloud. In *Conf on Hot Topics in Cloud Computing*, HotCloud'09.

Kenny, C. and Pahl, C. (2003). Automated tutoring for a database skills training environment. In *ACM SIGCSE Symposium 2005*, 58-64.

Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

Kiss, P., Reale, A., Ferrari, C. J., and Istenes, Z. (2018). Deployment of iot applications on 5g edge. In *2018 IEEE International Conference on Future IoT Technologies*.

Kritikos, K. and Skrzypek, P. (2018). A review of serverless frameworks. In *2018 IEEE/ACM UCC Companion*.

Lama, P. and Zhou, X. (2010). Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee. In *Intl Symp on Modeling, Analysis and Simulation of Computer and Telecom Systems*.

Le, V. T., Pahl, C. and El Ioini, N. (2019). Blockchain Based Service Continuity in Mobile Edge Computing. In *6th International Conference on Internet of Things: Systems, Management and Security*.

Lei, X., Pahl, C. and Donnellan, D. (2003). An evaluation technique for content interaction in web-based teaching and learning environments. In *3rd IEEE Intl Conf on Advanced Technologies*, 294-295.

Li, R., Zhao, Z., Zhou, X., Ding, G., Chen, Y., Wang, Z., and Zhang, H. (2017). Intelligent 5g: When cellular networks meet artificial intelligence. *IEEE Wireless Communications*, 24(5):175–183.

Lin, C.-T. and Lee, C. S. G. (1991). Neural-network-based fuzzy logic control and decision system. *IEEE Transactions on Computers*, 40(12):1320–1336.

Melia, M. and Pahl, C. (2009). Constraint-based validation of adaptive e-learning courseware. In *IEEE Transactions on Learning Technologies* 2(1), 37-49.

Mendonca, N. C., Jamshidi, P., Garlan, D., and Pahl, C. (2019). Developing self-adaptive microservice systems: Challenges and directions. *IEEE Software*.

Murray, S., Ryan, J. and Pahl, C. (2003). A tool-mediated cognitive apprenticeship approach for a computer engineering course. In *Proceedings 3rd IEEE International Conference on Advanced Technologies*, 2-6.

openFaaS (2019). openfaas: Auto-scaling. https://docs.openfaas.com/architecture/autoscaling/. Accessed: 2019-11-11.

Pahl, C., Barrett, R. and Kenny, C. (2004). Supporting active database learning and training through interactive multimedia. In *ACM SIGCSE Bulletin* 36 (3), 27-31.

Pahl, C., El Ioini, N., Helmer, S. and Lee, B. (2018). An architecture pattern for trusted orchestration in IoT edge clouds. Intl Conf Fog and Mobile Edge Computing.

Pahl, C., Jamshidi, P. and Zimmermann, O. (2018). Architectural principles for cloud software. ACM Transactions on Internet Technology (TOIT) 18 (2), 17.

Pahl, C. (2003). An ontology for software component matching. International Conference on Fundamental Approaches to Software Engineering, 6-21.

Pahl, C., Fronza, I., El Ioini, N. and Barzegar, H. R. (2019). A Review of Architectural Principles and Patterns for Distributed Mobile Information Systems. In *14th Intl Conf on Web Information Systems and Technologies*.

Pahl, C. (2005). Layered ontological modelling for web service-oriented model-driven architecture. In *Europ Conf on Model Driven Architecture – Found and Appl*.

Pahl, C., Jamshidi, P. and Zimmermann, O. (2020). Microservices and Containers. Software Engineering SE'2020.

Saboori, A., Jiang, G., and Chen, H. (2008). Autotuning configurations in distributed systems for performance improvements using evolutionary strategies. In *Intl Conf on Distributed Computing Systems*.

Samir, A. and Pahl, C. (2019). Anomaly Detection and Analysis for Clustered Cloud Computing Reliability. In *Intl Conf on Cloud Computing, GRIDs, and Virtualization*, 110–119.

Samir, A. and Pahl, C. (2019). A Controller Architecture for Anomaly Detection, Root Cause Analysis and Self-Adaptation for Cluster Architectures. In *Intl Conf on Adaptive and Self-Adaptive Syst and Appl*, 75–83.

Samir, A. and Pahl, C. (2020). Detecting and Localizing Anomalies in Container Clusters Using Markov Models. Electronics 9 (1), 64.

Scolati, R., Fronza, I., Ioini, N. E., Samir, A., and Pahl, C. (2019). A containerized big data streaming architecture for edge cloud computing on clustered single-board devices. In *Intl Conf on Cloud Computing and Services Science CLOSER*.

Steffenel, L., Schwertner Char, A., and da Silva Alves, B. (2019). A containerized tool to deploy scientific applications over soc-based systems: The case of meteorological forecasting with wrf. In *CLOSER 2019*.

Taibi, D., Lenarduzzi, V. and Pahl, C. (2019). Microservices Anti-Patterns: A Taxonomy. Microservices - Science and Engineering, Springer.

Taibi, D., Lenarduzzi, V., Pahl, C. and Janes, A. (2017). Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. In *XP2017 Scientific Workshops*.

von Leon, D., Miori, L., Sanin, J., El Ioini, N., Helmer, S. and Pahl, C. (2018). A Performance Exploration of Architectural Options for a Middleware for Decentralised Lightweight Edge Cloud Architectures. Intl Conf Internet of Things, Big Data & Security.

von Leon, D., Miori, L., Sanin, J., El Ioini, N., Helmer, S. and Pahl, C. (2019). A Lightweight Container Middleware for Edge Cloud Architectures. Fog and Edge Computing: Principles and Paradigms, 145-170.

Warden, P. (2019). Tensorflow 1.9 officially supports the raspberry pi. https://medium.com/tensorflow/tensorflow-1-9-officially-supports-the\-raspberry-pi-b91669b0aa0.

Xi, B., Xia, C. H., Liu, Z., Zhang, L., and Raghavachari, M. (2004). A smart hill-climbing algorithm for application server configuration. In *13th Int. Conf. on WWW*.