Fuzzy Container Orchestration for Self-adaptive Edge Architectures

Fabian Gand, Ilenia Fronza, Nabil El Ioini, Hamid R. Barzegar, Shelernaz Azimi, and Claus Pahl

Free University of Bozen-Bolzano, Bolzano, Italy
{fabian.gand, ilenia.fronza, nabil.ioini, hamidr.barzegar, shelernaz.azimi, claus.pahl}@unibz.it

Abstract. The edge or edge computing refers to the computational infrastructure between sensors and Internet-of-Things world on the one side and centralised cloud data centres on the other. Edge clusters could consist of small single-board devices, which are widely used in different applications. This includes microcontrollers regulating an industrial process or controllers monitoring and managing traffic roadside. Although generally hardware capabilities of devices are growing, resources at the edge are often still limited, requiring an intelligent resource management. This can be done through self-adaptive scaling mechanisms in these clusters allowing to scale components of the application in the cluster. We introduce here an auto-scalable cluster architecture for lightweight container-based edge devices. A serverless architecture forms the core of this architecture. The auto-scaling component is based on fuzzy logic in order to better manage uncertainty problems arising in such contexts. Our evaluations show that our platform architecture and the auto-scaling functionality satisfy the need of lightweightness in edge architecture.

Keywords: Edge cloud · Container technology · Cluster · Self-adaptive system · Performance engineering · Auto-scaling · Fuzzy logic

1 Introduction

Today, many processing tasks in distributed environments are executed directly on nodes at the edge of a network rather than being sent to central, but remote processing hubs. This principle is often called Edge Computing [52]. The management of such edge systems often requires different concepts for scaling system components dynamically in order to deal with changing resource requirements in a constrained environment. Proposed solutions range from simple algorithms that define scale values based on set thresholds [76] to systems using neural networks for decision making [42]. However, more work in the edge context [18] is needed on auto-scaling algorithms.

Our objective is the evaluation of a serverless architecture managed by an auto-scaling component. The goal is to provide a lightweight edge computing
solution, which extends our previous work in [18] by focusing more on the architecture implementation here. At the core is a serverless approach that delegates the deploying, scaling and maintaining of software to the cloud/edge provider [6]. The experimental edge platform we use here is implemented using a cluster of eight single-board devices. Such small clusters have been evaluated in basic settings. We aim here to evaluate a complete system based on real-life requirements and constraints with dynamic scalability included that would for instance arise in automotive and mobility applications.

We deploy Raspberry Pis as single-board devices and evaluate under which conditions a cluster of these devices can support low latency needs, which are tightly constrained by given performance requirements. We use technologies and tools such as MQTT for inter-cluster communication, openFaas and Docker Swarm for the implementation of the serverless concept, also Prometheus for monitoring purposes. This is combined with fuzzy logic at the core the central auto-scaling component. We look at the performance of the system in a range of cases, aiming to detect bottlenecks. The implemented auto-scaling algorithm is evaluated experimentally.

In the remainder of this paper, relevant concepts, tools and technologies are introduced in Sect. 2. Then, Sect. 3 starts with a high-level architecture, followed by details of the auto-scaling mechanism in Sect. 4. In Sect. 5, we evaluate platform and auto-scaler. Then, related work on distributed edge systems and auto-scaling is reviewed, before concluding with a summary and ideas for future research.

2 Background Concepts and Technologies

Our contribution is a scalable platform architecture, for which central concepts and technologies shall be introduced first [54,56].

2.1 Platform and Architecture Concepts

Our platform architecture requires some background in architecture, deployment and self-adaptivity [13,30,35,53].

Edge Computing. Cloud computing is based on the idea of centralised data centers that can process large volumes of data [20,24,25,39,43]. With edge computing, the potential local processing power at the “Edge” network is meant to be used, remedying the often significantly increased latency of a centralised approach. Edge Computing leverages the processing power of local edge nodes. These nodes are an intermediate layer [39] that process data within the local network rather than handling this in the cloud remotely.
**Microservice Architecture.** Microservices is a recent architectural paradigm. Traditionally, architectures provide 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 [32,58,69].

Usually, there needs to be some kind of bus system that the services use for communication. Consequently, each microservice needs to expose an interface for receiving and sending data. Often, this is implemented using HTTP methods and resources. Fowler suggests that these services should be split according to business capabilities rather than separating them based on the technology layer [14]. The size of microservices is highly disputed and ranges from services that can be managed by a single person to services that are developed by a dozen developers. There are different approaches to microservice architecture. The Micro-SOA approach splits the monolith into services that are relatively small (a few hundred to a few thousand lines of code) and need to interact with each other frequently. The corresponding architecture is highly scalable, but the constant communication between the services results in a higher network latency [70]. The Distributed DDD (Domain-Driven-Design) Framework aims at providing microservices that are an order of a magnitude bigger than in the Mico-SOA approach and are separated based on logical boundaries. They should be able to process requests independently and therefore contain their own domain data. Important challenges that need to be addressed when implementing an architecture based on microservices are service discovery (enabling the services to find each other), load balancing (splitting the load among services) and routing (making sure the requests are relayed to the right service).

**Serverless Computing.** Serverless computing is an new concept for the deployment of cloud applications that has seen an increase in popularity since it was first introduced a couple of years ago [6]. It allows developers to focus 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 centrally managed [6]. Serverless computing is related to the Functions-as-a-Service (FaaS) approach. Here, small chunks of functionality are deployed in the cloud and scaled dynamically by the cloud provider [40]. 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 [12]. Managing and deploying these microservices is similar to serverless functions.

In addition to major cloud providers already offering serverless functionality as part of their cloud solutions, several open-source frameworks have been developed and released in recent years. These solutions usually involve having to self-host the serverless frameworks on own hardware instead of relying on hardware provided by third-party providers. Here, we will mainly focus on open source, self-hosted solutions.
Table 1. Comparison of open-source, Serverless frameworks [18].

| Framework              | Languages                      | Type                      | License  | Vendor monitoring | Key components          | Other features                      |
|------------------------|--------------------------------|---------------------------|----------|-------------------|--------------------------|-------------------------------------|
| openFaas               | C#, Dockerfile, Go, Java, NodeJS, PHP, Python, Ruby | Complete framework        | MIT      | Community driven  | Prometheus               | Container (Docker), Prometheus, gateway API, GUI |
| OpenWhisk              | NodeJS, Swift, Java, Go, Scala, Python, PHP, Ruby, Ballerina | Complete framework        | Apache License 2.0 | Apache | Kamon for system metrics, kafka events for user metrics | nginx Webserver, CouchDB, Kafka, Containers (Docker) |
| Kubeless [41]          | Python, NodeJS, Ruby, PHP, Go, .NET, Java, Ballerina | Complete framework        | Apache License 2.0 | Community driven | Prometheus               | Kubernetes (native)                 |
| Serverless (community version) [67] | Python, NodeJS, Java, Go, Scala, C# | CLI for building + deploying | MIT      | Serverless, Inc  | –                       | Deployment to providers: AWS, MS, Azure etc. |

Four open-source frameworks are compared in Table 1, which aids us in the selection of a suitable one.

2.2 Self-adaptiveness and Fuzziness Concepts

Self-adaptive systems [2,28,29] adapt their behavior dynamically 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 [48,62–64]. In practice, platforms such as Kubernetes container orchestration facilitate the deployment and management of microservice applications. However, Kubernetes natively only supports basic auto-scaling by changing the number of service instances automatically.

Fuzzy logic aims to bridge between machine and human reasoning. Computers traditionally work well for tasks that contain formal calculations. Human reasoning, however, is often 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 [21]. Thus, human reasoning is fuzzy. Uncertainty also arises in edge environments through incomplete or potentially incorrect or conflicting observations.

Fuzzy logic has the objective of addressing 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 so-called membership function is a function that represents a fuzzy set and decides to which degree an item belongs to a certain set. Fuzzification is a process that refers to input values
that are mapped to membership functions to derive the degree of membership in that set [46] – which is different from 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. Defuzzification is the final step, where a numerical output value is generated.

An example shall illustrate this. A sample goal might be to calculate the money that should 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. We illustrate the ingredients of a fuzzy system:

- Membership functions – The variables salary, expected expenses and the money that is suggested to be saved are visualized in Figs. 1a, 1b and 2. Each variable consists of three membership functions: low, high and medium.
These can represent, e.g., the degree to which a salary of 50 can be considered a low, high or average salary.

- **Rules** – The rules of the fuzzy system can be defined in the following way:

  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 Input** – Savings are calculated based on a salary of 82 and expected expenses of 51.4.
- **Defuzzyfication** – After the final defuzzyfication step, a savings recommendation of 15.03 results.

## 2.3 Platform Tools and Technologies

The above abstract principles shall be complemented by the introduction of concrete infrastructure and software technologies used in the platform implementation.

**Raspberry Pi.** A *Raspberry Pi* is a widely used single-board computer based on an ARM-processor. Its original purpose was to introduce school children to programming [8, 73, 74]. Due to its low price and various use cases it has found its way into more industrial IT projects. Since the start of the project in 2012 there have been four major iterations of the Raspberry Pi platform. The version 2 B models that are used in this cluster include a 900 MHz quad-core ARM Cortex-A7 CPU and 1 GB of RAM [60]. The Raspberry Pi Model 3 B comes with a Quad Core 1.2 GHz Broadcom BCM2837 64 bit CPU. All (model B) Raspberry Pi versions include several GPIO Pins, a 100 Base Ethernet Port four USB Ports and an SD-Card slot.

**Docker and Docker Swarm.** *Docker* is software for containerization. Containerization is a virtualization technology that, instead of virtualizing hardware, separates processes from each other by utilizing certain features of the Linux kernel. It has been successfully used on Raspberry Pis [65, 66]. Docker containers bundle an application along with all of its dependencies. Docker offers the ability to create, build and ship containers [11]. Compared to virtual machines, containers offer a far better use of the host resources while providing similar advantages of having a separate system. Images are the blueprints of docker containers. Each container is created from an image. The images, on the other hand, are built using Dockerfiles which describe the system to be constructed. Docker (specifically the Docker Engine) is based on a client-server architecture. The client communicates with the Docker daemon via a command-line interface. The docker daemon is in charge of managing the components and containers. Docker services represent the actual application logic of a container in production. Using services, a distributed web application could be split into one service.
for the front-end components, one for the database and another one for the content management system that is used to update the website. Docker Swarm is the cluster management tool that is integrated into the Docker Engine. Instead of running the given 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 is also possible.

**Hypriot OS and Related Tools.** Hypriot OS is an operating system based on Debian that is specifically tailored towards using Docker Containerization technology on ARM devices such as the Raspberry Pi [22]. The OS comes prepackaged with the latest Docker version ready to be used. Hypriot OS also comes bundled with a few additional tools that we will use.

*Clout Init* – Clout init is a light-weight approach to creating templates of an operating system [9]. This can be used to ensure that two nodes are identical clones of each other. The instances are configured in .yml configuration files. These files can be used to set, for example, the hostname of the node or to execute certain commands at boot time.

*Avahi* – Avahi is a tool for simplifying networking by allowing nodes in a network to address each other with their hostnames without having to configure static IP addresses or rely on DHCP or DNS services [3]. If the hostname of the master node has been set in the cloud-init configuration file, all worker nodes are able to directly connect to the master node by addressing it by its hostname.

**Ansible.** Ansible is a tool for automating a variety of tedious tasks in a cluster and cloud environments such as individual node configuration or application deployment [1]. The nodes are usually connected using SSH. Ansible usually uses SSH keys for authorization [1]. The individual nodes need to be defined in a host configuration file. After defining the nodes, Ansible can be used to execute instructions on all nodes synchronously. These instructions can either be raw commands for simple cases or “playbooks”, that contain a set of instructions for more complex tasks.

**MQTT.** MQTT is a network protocol that is primarily used for unreliable networks with limited bandwidth, thus it is suitable for Internet of Things applications. MQTT uses a publisher-subscriber approach. Clients establish a connection to a broker and subscribe to topics. Clients may also publish a message to a topic. When a message is published, the broker relays the message to all clients that are subscribed to the corresponding topic. If a message is flagged as a retained message it is kept by the broker after relaying it. Clients will receive the retained message as soon as they subscribe to the topic.

**Prometheus.** Prometheus is a monitoring tool used to gather and process application metrics. In contrast to other monitoring tools, it does not rely on the
application delivering the metrics to the monitoring tool. Prometheus “scrapes” the metrics from a predetermined interface in a given interval. This means that the metrics are expected to be exposed by the application.

**OpenFaaS.** As a FaaS Functions-as-a-Service framework for containers, OpenFaaS can be deployed on top of a Docker swarm or a Kubernetes cluster. When starting the openFaas framework a number of standard docker containers are deployed:

- Gateway: used as the central gateway for calling functions from anywhere in the cluster. Exposes a webinterface for managing functions.
- Prometheus: a simple Prometheus instance is running on this container. In addition, the Prometheus webinterface is also exposed.
- Alertmanager: reads Prometheus metrics and issues alerts to the gateway.

openFaas does provide a simple form of autoscaling [50] that leverages the default metrics aggregated by Prometheus and scales based on given thresholds.

Functions and serverless microservices can be written in a variety of different programming languages. Functions are usually created by using the faas-cli command line tool. An example for creating a function in python would be:

```
faas-cli new --prefix=examples --lang python3-armhf example-function
```

This command creates a directory structure as follows:

```
example-function
  example-function
    handler.py
    requirements.txt
  example-function.yml
```

The `.yml` configuration file contains information such as the IP address of the gateway or the name of the docker image. The programming logic can be found

---

**Fig. 3.** Hardware and networking components [18].
within the `handler.py` file. It needs to contain a `handle` function that receives a request parameter and returns a response after executing its logic. The `requirements.txt` file contains the python-pip dependencies needed for the project. The corresponding pip install command is executed automatically once the function is built. The `--lang` parameter determines the language and architecture template that is used for this function. In the given case, a python function running on an ARM architecture is created. The `--prefix` parameter is used as a prefix for the resulting Docker image that is uploaded to the Docker hub. The Docker hub is used for distributing the images across the cluster. Stateless microservices can be built by using the `Dockerfile` template. In this case a Dockerfile has to be set up manually to create a container for the service to run in. Among the serverless frameworks compared in Table 1, we selected openFaas for the implementation of the application. The reasons are a wide array of supported languages, openFaas being a complete, all-in-one framework, Prometheus as an integrated, extendable monitoring solution, the simple set-up process and its out-of-the-box support for Docker Swarm. Other frameworks either lacked a simple monitoring solution (serverless), included more custom components that needed to be configured manually (openWhisk) or did not support Docker Swarm (Kubeless).

2.4 Towards a Lightweight Edge Platform

The architecture approach is decomposing the application into microservices and containerizing them, by utilizing the above technologies and allowing the hardware to be reallocated dynamically.

OpenFaas allows building and deploying services in the form of functions across a cluster. Some of its features, such as built-in Prometheus services or the gateway, can be extended to make openFaas the central building block [19]. An overview of the different technologies in the context of the given platform is presented in Fig.3.

OpenFaas also enables scaling the different parts of the application. Even though openFaas scaling is limited, we use it as a foundation for our self-adaption. We implement a more fine-grained scaling algorithm using the built-in monitoring options. As the reasoning foundation for scaling, we use fuzzy logic.

3 Platform and Controller Architecture

We introduce here the core elements of our architecture before covering low-level implementation details of the platform and its auto-scaling component in the next section. 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 reconfiguring a few parameters.

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. A control system is used to scale the TM System based on the monitored and analysed data. While we investigate a specific case, transferability is given.

![Fig. 4. Interaction between different systems [18].](image1)

We build on a three-layered architecture. 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 and controller layer and additional components.

![Fig. 5. Platform layer [18].](image2)

### 3.1 Platform and Cluster Configuration

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. These elements of the platform layer are shown in Fig. 5.

**Cluster Setup.** Our cluster architecture 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). We have documented our solution on Github. The system components are split into three repositories. The rpicluster repository\(^1\) contains the clout-init configuration files for setting up the Raspberry Pis. The rpicluster-application repository\(^2\) includes all the microservices and scripts that make up the application logic. The third repository\(^3\) includes the openFaas repository in a modified form.

All nodes of the cluster run hypriot OS. Cloud-init is used to define the initial boot steps of the nodes. The master initiates the docker swarm. The only command that needs to be executed on the workers is the swarm join command. It can be distributed among the nodes by using Ansible. After this command is executed on each node, the swarm is fully set up. The worker nodes contain almost no additional dependencies since they are all included in the Docker containers. The only additional dependencies that are directly installed on the nodes are used to run a python script that monitors system metrics and publishes them to the metrics service.

The complete steps to set-up the system on the cluster are as follows:

1. Flash one master configuration of hypriot to an SD Card and insert it into the master RPI:

   ```bash
   flash --hostname master --userdata ./user-data-master.yml hypriotos.img
   ```

2. Flash \(n\) worker configurations of hypriot to SD Cards and insert them into worker RPIs.
3. Power up the cluster.
4. Wait for the predefined boot steps to download and set up the system. Logs can be inspected on the RPI nodes at: `/var/logs/cloud-init-output.log`
5. On the master node:
   a. Add the hostname of all workers to the `hosts` file in the root directory.
   b. Use the ansible command line tool to execute the docker join command on all workers.
   c. See if all nodes joined the swarm of the cluster: `docker node ls`
6. Git clone the modified openFaas repository:

\(^1\) [https://gitlab.com/gandfabian/rpicluster.git](https://gitlab.com/gandfabian/rpicluster.git).
\(^2\) [https://gitlab.com/gandfabian/rpicluster-application.git](https://gitlab.com/gandfabian/rpicluster-application.git).
\(^3\) [https://gitlab.com/gandfabian/rpicluster-faas.git](https://gitlab.com/gandfabian/rpicluster-faas.git).
7. Execute the openFaas setup script:

```
~/rpicluster-faas/deploy_stack.armhf.sh
```

8. Wait until the built-in openFaas services are started (can be checked by calling `docker service ls`). After startup is complete, the openFaas GUI (at http://master:8080) and the modified Prometheus, instances configured for use with custom metrics, (at http://master:9090) are accessible from the network.

The system can now be started from within the `rpicluster-application` folder by executing the `control.py` script that will use openFaas to start-up all application components and enable auto-scaling.

### 3.2 Platform Monitoring

Monitoring is done through the openFaas Prometheus instance, provides as one of the predefined containers.

**Metrics Service.** The metrics service is used to acquire metrics about the system, mainly by serving as a central hub that accumulates all cluster-wide metrics and by publishing those metrics via a flask HTTP endpoint. This endpoint is the central interface for Prometheus to scrape from. The Prometheus python API is used to implement the metrics. The metrics service exposes the number of messages, e.g., the number of active cars in our case as well as the cumulative memory and CPU usage. The number of messages is implemented as a counter which is continuously increasing. The CPU/memory usage, on the other hand, is realized by using a gauge which can be a set to an arbitrary number.

**OpenFaas: Prometheus.** This instance is used to store metrics and query them when needed. Prometheus provides a REST API along with a language called PromQL to aggregate and query metrics [59]. The aggregated data is returned in the JSON format. Before startup, Prometheus needs to be informed about the endpoints that metrics should be collected from. The Prometheus instance that is shipped with openFaas only scrapes metrics from the openFaas gateway since only metrics related to function execution are being monitored by default. Configuring the openFaas Prometheus instance to aggregate custom metrics of an application is not documented. Exploring this possibility and implementing it was a part of the scope of this thesis. In order to add a second
endpoint for the additional metrics, the openFaas repository had to be forked and the prometheus/prometheus.yml configuration file had to be edited, adding the metrics endpoint to the file. The metrics microservice is accessible via the gateway. Therefore, it is possible to address the metrics endpoint by calling the gateway and there is no need to specify the static IP address of the node the metrics service is running on. Specifying the metrics path (/appmetrics) as well as the port (8080) is also mandatory.

4 A Fuzzy Auto-scaling Controller

The central component in our architecture that manages performance is the auto-scaling controller. A requirement for it is lightweightness. Consuming too many system resources such as storage space or CPU usage has to be avoided because the algorithm is meant to be deployed on the RPI cluster itself with most of the clusters resources being reserved for application components themselves.

At the core is fuzzy logic that is a good compromise between a powerful decision-making process and a limited resource consumption. The initial fuzzy knowledge base is, however, 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).

In order to set-up an initial fuzzy knowledge base, values of previous runs of the system are used. The reactive part of the algorithm continuously updates parts of the knowledge base dynamically.

![Fig. 6. MAPE-K loop for fuzzy auto-scaling controller [18].](image-url)
4.1 Principles of Auto-scaling

The auto-scaling algorithm builds on the MAPE-K \[38\] controller loop for self-adaptive systems. The steps of this MAPE-K controller loop are 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, continuously updated knowledge base.

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

Continuous Scaling: The reactive part of the algorithm is executed as the default after start. It aims to adjust the 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 membership functions. The goal is to constantly update the membership function such that the fuzzy service provides optimal scale values for all load scenarios. 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 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. The definitions of the membership functions are also part of the knowledge component and are continuously updated (Fig. 7).

4.2 The Fuzzy Service

The task of the fuzzy service is to determine the scale value. Its rules are predefined. It uses the membership functions as input. Another parameter is the range of the scale values, which can be defined to set the minimum and maximum scale values that are considered acceptable. The service returns a global scale value as output.

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,
Fig. 7. Details of services providing scaling functionality [18].

(a) Initial membership functions.

(b) Re-adjusted membership functions: the system can handle less messages than expected.

Fig. 8. Scaling service [18].

the allowed number of replicas will decrease. The minimal alpine distribution, which is used for the other images, could not be used in this case since sk-fuzzy, 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.

4.3 Controller Calibration and Configuration

The metrics that are gathered during earlier system runs are used as the basis for calibrating the fuzzy membership functions in order to manage the anticipated demand better. The metric used here 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 has to process since monitoring started. This requires a way to measure how many messages are processed in a given time frame. This is done using the Prometheus rate() function, which returns the per second rate of increase.

The following example query is used throughout the application to return the per-second rate of increase measured for the past 20 s:

\[
\text{rate(messages\_count\_total[20s])}
\]

The increase rate \( r \) is then calculated as follows:

\[
r = \frac{x_i - x_{i-t}}{t}
\] (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 \( \sigma_g \) as their own standard deviation. The mean of the average membership function is placed at the global median:

\[
\text{median}(a)_g
\] (2)

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

\[
\text{median}(a)_g - \sigma_g
\] (3)

with \( \sigma_g \) again being the global standard deviation. The mean of the high membership function is thus:

\[
\text{median}(a)_g + \sigma_g
\] (4)

The initial membership functions are shown in Fig. 8a.

The next step is the calculation of the three global variables. As an example, the following query provides the median value over all rates of messages (i.e., the rates covering a time span of 20 s each) of the last 90 days:

\[
\text{quantile\_over\_time(}
0.5, \text{rate(messages\_count\_total[20s])[90d:20s] )}
\]

While Prometheus does not support a median function, the built-in quantile function can be used to calculate the median. A quantile splits a probability
distribution into groups according to a given threshold. The 0.5 quantile equals the median. Generally, the median can be defined as:

$$Q_A(0.5) = \text{median}(a)$$  \hspace{1cm} (5)

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

```promql
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 20s each). The total time span that is considered are the last 90 days. The standard deviation for the given context is:

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

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

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

### 4.4 Continuous Auto-scaling

Auto-scaling is based on the previous initial membership functions that are re-adjusted 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 that scales the relevant services:

$$s = \text{round}(D \times G)$$  \hspace{1cm} (7)

Here, $s$ denotes the new scale value that a service is scaled to. $D$ refers to 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. The scalable components include the gatherer service and the decision function that were introduced earlier.

The Message Roundtrip Time MRT is calculated after a scaling action. 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. 8a 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. 8b. 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 adjusted (shifted) in each iteration of the loop can be controlled by (manually) updating the adjustment factor. Hence, the membership functions for each iteration are determined in the following way:

- Membership function low: \( \text{median}(a) - \sigma + A \)
- Membership function average: \( \text{median}(a) + A \)
- Membership function high: \( \text{median}(a) + \sigma + A \)

Here, \( A \) is the adjustment factor for the functions. \( A \) is computed in each iteration. A positive re-adjustment factor results in a shift to the right, a negative one 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 with the control script, the reactive part of the algorithm is started again, providing the fuzzy service with the re-adjusted membership functions.

Algorithm 1 outlines the continuous reactive scaling functionality explained above.

**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)
5 Controller Evaluation

Performance is the critical property. Thus, the evaluation shall focus on the performance of the proposed serverless architecture with the auto-scaling mechanism at the core and determine crucial bottlenecks.

5.1 Evaluation Objectives and Metrics

The auto-scaling approach needs to be evaluated in terms of its effectiveness 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 also determined the maximum load, i.e., here the maximum number of vehicles the architecture (including the network it is operated in) can support.

The evaluation goals were analyzed 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. As a second step, the evaluation procedure was repeated for a complete cluster (consisting of eight RPis).

For the evaluation, we report a range 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 20s after the previous scaling operation was completed. Here, the maximum scale value was unknown. In concrete scenarios, this value could be determined prior to execution. 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.

5.2 Evaluation Setup

Some configurations were made to facilitate the evaluation. 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.

\[^4\text{Note, that the vehicles referred to here could be replaced by other application components.}\]
Table 2. Results for cluster of 3 RPIs. Scaling was active with variables set to values reported in Fig. 3 [18].

| 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 3. Initial Calibration Scaling run for cluster of three RPIs with fixed number of 12 cars. Shift Left/Right reports on the direction the membership functions are shifted during an iteration. The number of scaled components is also reported. Manually set variables: Maximum Scale Value: 5, MRT Threshold: 2.0 s – FSV: Fuzzy Scale Value, IT: Invocation Time after scaling, Ga: Gatherer, DF: Decision Function [18].

| 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    |

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. Prometheus then processes these.

5.3 Evaluation Implementation

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 2 reports on the initial set of metrics for different numbers of vehicles. The scaling functionality was active when the metrics were monitored. Table 3 includes data of the scaling algorithm for an initial calibrating run.
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 4.

Table 4. Scaling experiment for cluster of 8 RPIs. Manually set variables: Maximum Scale Value: 12, Number of cars: changing, MRT Threshold: 0.3 s – FSV: Fuzzy Scale Value, AF: Adjustment Factor [18].

| 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 |

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 initial setup, however, shows a MRT of about 1.6 s for only 2 vehicles. At 12 vehicles, a Roundtrip Time of 4.5 s is reached and at 14 vehicles the MRT is already above 10 s, 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 s. 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 s. In the second iteration, the MRT value is recorded above the threshold at 2.5 s. 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 drops below the threshold again.

5.4 Discussion of Results

We derived starting values for all variables using the calibration pilot on a smaller cluster set-up. The evaluation of the architecture indicated that serverless function calls should be restricted since they introduce network latency problems.
The bottleneck is here the network. The given 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 extensions should find different network set-ups that allow for a dependable solution overcoming the limits identified here.

Our fuzzy auto-scaling solution works as intended, i.e., it is able to scale the system in a balanced manner. After finding a SLO, usually given in the system specifications, only the maximum scale value and the adjustment factor are the values that need to be set manually.

We can conclude that the full version of our system architecture provides satisfactory results in terms of resource consumption and performance (MRT). Furthermore, we could demonstrate acceptable scalability of the system.

6 Related Work

The discussion of serverless technology in academia is still in its early beginnings as noted by Baldini et al. [6]. This is a stark contrast to the attention it has gotten from industry and the software engineering community. Baldini et al. go on to introduce several commercial implementations such as Amazon’s AWS Lambda or Microsoft Azure Functions. The major advantages mentioned are the responsibility of managing servers and infrastructure that no longer rests with the developer. Another advantage is that the provider may offer additional services such as logging or monitoring to be used on top of the client’s applications. Kritikos et al. review several serverless framework and report on challenges faced when designing and implementing an application leveraging serverless technology [40]. They report on the need for new methods for creating an architecture for applications that contain both serverless components like functions as well as “classic” components such as microservices running inside a Docker Container [71]. They also note that the decision on how the application components should scale is largely left to the developer and suggest further research into automating this process. Lastly, they add that monitoring could be improved. Especially the option to use and aggregate custom metrics is noted as “missing in most of the frameworks” [40].

Kiss et al. gather requirements for applications making use of Edge Computing [39]. Specifically in the context of combining them with the capabilities of the 5G standard. They mention that recently released single-board devices open up the possibility of processing some of the data at the edge of the cluster. This, however, goes along with the challenge of orchestrating the available processing power. The IoT system needs to be able to “reorganize itself” [39] based on changing conditions. In order to create such a system, they propose the following steps: In the discovery phase the granular parts of the system are identified and the application is split up accordingly. Based on the observations, a deployment plan is created. This is followed by the execution phase in which the parts of the application are deployed to their corresponding locations and the system is
started. The system’s performance is monitored at run-time and reconfiguration steps are taken if necessary. The cycle is concluded by the learning and predicting step. To make use of the data gathered at run-time, the system needs to be able to learn using AI from it in order to improve its reconfiguration steps.

There different examples of comparable IoT Systems that have been documented. In order to be comparable to the proposed solution in this thesis, the systems had to be comprised of different hardware nodes that needed to communicate with each other as well as with a central node. Tata et al. outline the state of the art as well as the challenges in modeling the application architecture of IoT Edge Applications [72]. One of the exemplary scenarios they introduce is a system for smart trains based on the principles of edge computing. The system is comprised of a set of different sensors attached to crucial components of a car. The data is gathered by a central unit for each car that sends its own data to the central processing unit of the train. This unit is then able to communicate with the cloud. The task of gathering and processing data is shifted to nodes at the edge of the network before passing it on the cloud. They go on to suggest further research in the field of finding the optimal balance between modeling an IoT application independently of its environment and designing it in a way that accurately represents the structure of the environment it is deployed in. Another issue they mention is the scalability of such applications since the number of cars and sensors on different trains can vary significantly. The modeling of the architecture should therefore consider a way to deploy some parts of the application only to a specific set of nodes.

The mentioned papers offer an overview over the requirements and create proposals for the architecture of distributed IoT Systems. They also offer guidance on where more research could be conducted. They remain, however, on an abstract level and do not implement a prototype of a corresponding application. The work presented in this paper aims at making use of the proposed approaches to implement a concrete, distributed IoT System based on a real-life scenario that is executable, observable and analyzable. The task of leveraging custom application metrics to enable auto-configuration will also be addressed. The devised solution will be evaluated in a second step.

A system similar to ours in architectural terms as a Raspberry Pi-based implementation of an application system has been introduced in [68]. 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 scalability approaches. [31] 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 can be used to improve the network response to certain environmental factors such as network traffic or threats to the system [45]. The authors propose a general architecture of smart 5G system that includes an independent AI controller that communicates with the components of the application. The controller obtains certain metrics from these interfaces, uses its own functionality (based on a cycle of sensing, mining, prediction and reasoning) to analyse results and sends these back to the system while adapting its own parameters dynamically to improve its accuracy. The paper categorizes machine learning, the subset of AI it focuses on, into three categories:

Examples of implementations of algorithms for auto-scaling and auto-configuration exist. [61] 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. This algorithm does not rely on pre-existing information about the system and is therefore a black-box approach. The aim of the algorithm is to find an optimal set of parameters to “maximize throughput while keeping the response time below a threshold” [61]. This threshold is defined in a so called Service-Level-Agreement (SLA) [7,55,57].

Another paper introduces a Smart Hill Climbing Algorithm for finding the optimal configuration for a Web application [76]. 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 [36]. This is an example of an offline, proactive tuning algorithm that does not reallocate resources at run-time, but tries to find the best configuration in advance. The proposed algorithm starts by computing a signature of the resource consumption. This signature is then compared against a database of signatures of other applications. This database maps the signature of the different applications to the predefined set of optimal parameters. The configuration parameters of the current application are then updated with the values of the application from the database that most closely resembles.

There are some proposals using fuzzy logic for auto-configuration. [31] 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 [26,33]. 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 [26]. The idea of using supervised learning approaches and relying on training data has also been discussed. Yigitbasi et al. introduce an approach based on the Support-Vector-Regression Model [77]: a set of several machine-learning algorithms based on training-data that are used for classification and regression. Fuller introduces the idea of using Neural Networks for building and improving the fuzzy knowledge base [17]. Adaptive neuro-fuzzy inference systems (ANFIS), combining neural networks and fuzzy logic, are discussed in [34]. He states that neural networks excel at pattern recognition but it is usually difficult to follow their “reasoning” process. This process is easier to comprehend for fuzzy logic systems but they rely on a knowledge base that is not trivial to obtain. Fuzzy logic and neural networks are shown to complement each other in a hybrid system. Newer papers highlight a “self-adaptive neural fuzzy controller” for server provisioning [42]. The proposed architecture is making use of a neural network of four layers. The layers of the neural network represent the fuzzy membership function and rule base. The neural network will constantly learn and adapt itself. This will lead to the membership functions and rules being established and refined over time. It excels over a regular fuzzy controller based on manual tuning of the knowledge base.

Our work here focused on introducing a practical implementation approach for a lightweight auto-scaling controller based on fuzzy logic. So far, fuzzy auto-scaling for lightweight edge architectures has not been investigated in the literature.

7 Conclusions

At the core of our platform is a containerized architecture on a cluster of single-board devices. Combined with a fuzzy auto-scaler, this results overall in a reconfigurable, scalable and dependable system. 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 carried out to analyze the performance. OpenFaas was used for inter-service communication as well as for monitoring. The auto-scaling algorithm was specifically designed to support dependable resource management for lightweight edge device clusters.

Using fuzzy logic, the auto-scaling controller can gradually scale the system. Previous examples of fuzzy auto-scalers that were deployed in large cloud infrastructures, the fuzzy scaling functionality in our case was constrained in its processing needs since it was deployed together with 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. This also applies for fuzzy systems based on neural networks. There are still 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.

In our use case scenarios. Our evaluation demonstrated that the set-up is only able to process up to 75 vehicles simultaneously. While our focus was on compute/storage resources, here network limitations emerge as the reason. Exploring these more deeply is part of future work. We shall also focus on improving the application management components towards more realistic behavior.

We discussed related work on Adaptive Neuro-Fuzzy Inference Systems (called ANFIS). Working examples of ANFIS are relatively uncommon. One example uses python and the machine-learning framework tensorflow to combine fuzzy logic and a neural network [10]. 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 [75]. 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: \((rate \ of \ messages, \ scale \ value) \rightarrow \text{acceptable/not acceptable performance}\). This dataset could be aggregated manually or in an automated manner to address the problems.

We also plan to address more complex architectures with multiple clusters to be coordinated [15,16]. We propose Particle Swarm Optimization (PSO) [4] here, which is a bio-inspired optimization methods suitable to coordinate between autonomous entities such as clusters in our case, combined with machine learning techniques for the construction of the controller [5]. 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.

Our abstract application scenarios, with cars just being simulated data providers, allows us to generalise the results beyond the concrete application case [23,27]. We considered 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 [37,44,47,49,51]. These systems also rely on close interaction with semantic processing of interactions in order to support cognitive learning processes. Some of these can be provided at edge layer to enable satisfactory user experience based on sufficient platform performance.

References

1. Ansible: Overview: How ansible works (2020). https://www.ansible.com/overview/how-ansible-works. Accessed 04 June 2020
2. Ardagna, C.A., Asal, R., Damiani, E., Dimitrakos, T., El Ioini, N., Pahl, C.: Certification-based cloud adaptation. IEEE Trans. Serv. Comput. 14, 82–96 (2018)
3. Avahi - What is avahi? (2020). https://www.avahi.org/. Accessed 04 June 2020
4. Azimi, S., Pahl, C., Shirvani, M.H.: Particle swarm optimization for managing performance in multi-cluster IoT edge architectures (2020)
5. Azimi, S., Pahl, C.: Root cause analysis and remediation for quality and value improvement in machine learning driven information models. In: 22nd International Conference on Enterprise Information Systems ICEIS (2020)
6. Baldini, I., et al.: Serverless computing: current trends and open problems. CoRR, abs/1706.03178 (2017)
7. Barrett, R., Patcas, L.M., Pahl, C., Murphy, J.: Model driven distribution pattern design for dynamic web service compositions. In: Proceedings of the 6th International Conference on Web Engineering, pp. 129–136 (2006)
8. Belam, M.: The raspberry Pi: reviving the lost art of children’s computer programming (2012). https://www.theguardian.com/commentisfree/2012/feb/29/raspberry-pi-childrens-programming. Accessed 04 June 2020
9. Cloud-Init: cloud-init: The standard for customising cloud instances (2020). https://cloud-init.io/. Accessed 04 June 2020
10. Cuervo, T.: TensorANFIS (2020). https://github.com/tiagoCuervo/TensorANFIS. Accessed 04 June 2020
11. Docker: What is docker? (2020). https://www.redhat.com/en/topics/containers/what-is-docker. Accessed 04 June 2020
12. Ellis, A.: Introducing stateless microservices for openfaas (2018). https://www.openfaas.com/blog/stateless-microservices/
13. Fang, D., Liu, X., Romdhani, I., Jamshidi, P., Pahl, C.: An agility-oriented and fuzziness-embedded semantic model for collaborative cloud service search, retrieval and recommendation. Future Gener. Comput. Syst. 56, 11–26 (2016)
14. Fowler, M.: Microservices (2014). https://martinfowler.com/articles/microservices.html. Accessed 04 June 2020
15. Fowley, F., Pahl, C., Jamshidi, P., Fang, D., Liu, X.: A classification and comparison framework for cloud service brokerage architectures. IEEE Trans. Cloud Comput. 6(2), 358–371 (2016)
16. Fowley, F., Pahl, C., Zhang, L.: A comparison framework and review of service brokerage solutions for cloud architectures. In: Lomuscio, A.R., Nepal, S., Patrizi, F., Benatallah, B., Brandič, I. (eds.) ICSOC 2013. LNCS, vol. 8377, pp. 137–149. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06859-6_13
17. Fuller, R.: Neural Fuzzy Systems (1998)
18. Gand, F., Fronza, I., El Ioini, N., Barzegar, H.R., Azimi, S., Pahl, C.: A fuzzy controller for self-adaptive lightweight edge container orchestration. In: 10th International Conference on Cloud Computing and Services Science CLOSER (2020)
19. Gand, F., Fronza, I., El Ioini, N., Barzegar, H.R., Pahl, C.: Serverless container cluster management for lightweight edge clouds. In: 10th International Conference on Cloud Computing and Services Science CLOSER (2020)
20. Gand, F., Fronza, I., El Ioini, N., Barzegar, H.R., Pahl, C.: A lightweight virtualisation platform for cooperative, connected and automated mobility. In: 6th International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS) (2020)
21. Hong, T.-P., Lee, C.-Y.: Induction of fuzzy rules and membership functions from training examples. Fuzzy Sets Syst. 84(1), 33–47 (1996)
22. Hypriot: HypriotOS (2020). https://blog.hypriot.com/about/. Accessed 04 June 2020
23. El Ioini, N., Pahl, C.: A review of distributed ledger technologies (2018)
24. El Ioini, N., Pahl, C.: Trustworthy orchestration of container based edge computing using permissioned blockchain. In: International Conference on Internet of Things: Systems, Management and Security (IoTMS) (2018)
25. El Ioini, N., Pahl, C., Helmer, S.: A decision framework for blockchain platforms for IoT and edge computing. In: IoTBDs 2018 (2018)
26. Ipek, E., Mutlu, O., Martinez, J.F., Caruana, R.: Self-optimizing memory controllers: a reinforcement learning approach. In: 2008 International Symposium on Computer Architecture, pp. 39–50 (2008)
27. Jamshidi, P., Pahl, C., Chimenyeze, S., Liu, X.: Cloud migration patterns: a multi-cloud service architecture perspective. In: Toumani, F., et al. (eds.) ICSOC 2014. LNCS, vol. 8954, pp. 6–19. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22885-3_2
28. Jamshidi, P., Sharifloo, A., Pahl, C., Arabnejad, H., Metzger, A., Estrada, G.: Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. In: International Conference on Quality of Software Architectures, pp. 70–79 (2016)
29. Jamshidi, P., Pahl, C., Mendonca, N.C.: Managing uncertainty in autonomic cloud elasticity controllers. IEEE Cloud Comput. 3, 50–60 (2016)
30. Jamshidi, P., Pahl, C., Mendonca, N.C.: Pattern-based multi-cloud architecture migration. Softw. Pract. Exp. 47(9), 1159–1184 (2017)
31. Jamshidi, P., Ahmad, A., Pahl, C.: 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 (2014)
32. Jamshidi, P., Pahl, C., Mendonca, N.C., Lewis, J., Tilkov, S.: Microservices: the journey so far and challenges ahead. IEEE Softw. 35(3), 24–35 (2018)
33. Jamshidi, P., Sharifloo, A.M., Pahl, C., Metzger, A., Estrada, G.: Self-learning cloud controllers: fuzzy Q-learning for knowledge evolution. In: ICAC 2015, pp. 208–211 (2015)
34. Jang, J.S.R.: ANFIS: adaptive-network-based fuzzy inference system. IEEE Trans. Syst. Man Cybern. 23(3), 665–685 (1993)
35. Javed, M., Abgaz, Y.M., Pahl, C.: Ontology change management and identification of change patterns. J. Data Semant. 2(2–3), 119–143 (2013)
36. Kambatla, K., Pathak, A., Pucha, H.: Towards optimizing hadoop provisioning in the cloud. In: Conference on Hot Topics in Cloud Computing, HotCloud 2009 (2009)
37. Kenny, C., Pahl, C.: Automated tutoring for a database skills training environment. In: ACM SIGCSE Symposium 2005, pp. 58–64 (2005)
38. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
39. Kiss, P., Reale, A., Ferrari, C.J., Istenes, Z.: Deployment of IoT applications on 5G edge. In: 2018 IEEE International Conference on Future IoT Technologies (2018)
40. Kritikos, K., Skrzypek, P.: A review of serverless frameworks. In: 2018 IEEE/ACM UCC Companion (2018)
41. Kubeless: Kubeless: The kubernetes native serverless framework (2020). https://kubeless.io/. Accessed 04 June 2020
42. Lama, P., Zhou, X.: Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee. In: International Symposium on Modeling, Analysis and Simulation of Computer and Telecom Systems (2010)
43. Le, V.T., Pahl, C., El Ioini, N.: Blockchain based service continuity in mobile edge computing. In: 6th International Conference on Internet of Things: Systems, Management and Security (2019)

44. Lei, X., Pahl, C., Donnellan, D.: An evaluation technique for content interaction in web-based teaching and learning environments. In: 3rd IEEE International Conference on Advanced Technologies, pp. 294–295 (2003)

45. Li, R., et al.: Intelligent 5G: when cellular networks meet artificial intelligence. IEEE Wirel. Commun. 24(5), 175–183 (2017)

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

47. Melia, M., Pahl, C.: Constraint-based validation of adaptive e-learning courseware. IEEE Trans. Learn. Technol. 2(1), 37–49 (2009)

48. Mendonca, N.C., Jamshidi, P., Garlan, D., Pahl, C.: Developing self-adaptive microservice systems: challenges and directions. IEEE Softw. 38, 70–79 (2019)

49. Murray, S., Ryan, J., Pahl, C.: A tool-mediated cognitive apprenticeship approach for a computer engineering course. In: Proceedings 3rd IEEE International Conference on Advanced Technologies, pp2–6 (2003)

50. openFaaS: openfaas: Auto-scaling (2020). https://docs.openfaas.com/architecture/autoscaling/. Accessed 04 June 2020

51. Pahl, C., Barrett, R., Kenny, C.: Supporting active database learning and training through interactive multimedia. ACM SIGCSE Bull. 36(3), 27–31 (2004)

52. Pahl, C., El Ioini, N., Helmer, S., Lee, B.: An architecture pattern for trusted orchestration in IoT edge clouds. In: International Conference on Fog and Mobile Edge Computing (2018)

53. Pahl, C., Jamshidi, P., Zimmermann, O.: Architectural principles for cloud software. ACM Trans. Internet Technol. (TOIT) 18(2), 17 (2018)

54. Pahl, C.: An ontology for software component matching. In: Pezzè, M. (ed.) FASE 2003. LNCS, vol. 2621, pp. 6–21. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36578-8_2

55. Pahl, C., Giesecke, S., Hasselbring, W.: An ontology-based approach for modelling architectural styles. In: Oquendo, F. (ed.) ECSA 2007. LNCS, vol. 4758, pp. 60–75. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75132-8_6

56. Pahl, C., Fronza, I., El Ioini, N., Barzegar, H.R.: A review of architectural principles and patterns for distributed mobile information systems. In: 14th International Conference on Web Information Systems and Technologies (2019)

57. Pahl, C.: Layered ontological modelling for web service-oriented model-driven architecture. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 88–102. Springer, Heidelberg (2005). https://doi.org/10.1007/11581741_8

58. Pahl, C., Jamshidi, P., Zimmermann, O.: Microservices and containers. In: Software Engineering SE 2020 (2020)

59. Prometheus: Prometheus: Querying (2020). https://prometheus.io/docs/prometheus/latest/querying/. Accessed 04 June 2020

60. Raspberry Pi: Raspberry pi: products (2020). https://www.raspberrypi.org/products/. Accessed 04 June 2020

61. Saboori, A., Jiang, G., Chen, H.: Autotuning configurations in distributed systems for performance improvements using evolutionary strategies. In: International Conference on Distributed Computing Systems (2008)

62. Samir, A., Pahl, C.: Anomaly detection and analysis for clustered cloud computing reliability. In: International Conference on Cloud Computing, GRIDs, and Virtualization, pp. 110–119 (2019)
63. Samir, A., Pahl, C.: A controller architecture for anomaly detection, root cause analysis and self-adaptation for cluster architectures. In: International Conference on Adaptive and Self-Adaptive Systems and Applications, pp. 75–83 (2019)
64. Samir, A., Pahl, C.: Detecting and localizing anomalies in container clusters using Markov models. Electronics 9(1), 64 (2020)
65. Scolati, R., Fronza, I., Ioini, N.E., Samir, A., Pahl, C.: A containerized big data streaming architecture for edge cloud computing on clustered single-board devices. In: International Conference on Cloud Computing and Services Science CLOSER (2019)
66. Scolati, R., Fronza, I., El Ioini, N., Samir, A., Barzegar, H.R., Pahl, C.: A containerized edge cloud architecture for data stream processing. In: Ferguson, D., Méndez Muñoz, V., Pahl, C., Helfert, M. (eds.) CLOSER 2019. CCIS, vol. 1218, pp. 150–176. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-49432-2_8
67. Serverless: Serverless framework (2020). https://serverless.com/. Accessed 04 June 2020
68. Steffenel, L., Schwertner Char, A., da Silva Alves, B.: A containerized tool to deploy scientific applications over Soc-based systems: the case of meteorological forecasting with WRF. In: CLOSER 2019 (2019)
69. Taibi, D., Lenarduzzi, V., Pahl, C.: Microservices anti-patterns: a taxonomy. Microservices, pp. 111–128. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-31646-4_5
70. Taibi, D., El Ioini, N., Pahl, C., Janes, A.: Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. In: XP 2017 Scientific Workshops (2017)
71. Taibi, D., El Ioini, N., Pahl, C., Niederkofler J.R.S.: Patterns for serverless functions (Function-as-a-Service): a multivocal literature review. In: Proceedings of the 10th International Conference on Cloud Computing and Services Science, CLOSER (2020)
72. Tata, S., Jain, R., Ludwig, H., Gopisetty, S.: Living in the cloud or on the edge: opportunities and challenges of IoT application architecture. In: 2017 IEEE International Conference on Services Computing (SCC), pp. 220–224
73. von Leon, D., Miori, L., Sanin, J., El Ioini, N., Helmer, S., Pahl, C.: A Performance exploration of architectural options for a middleware for decentralised lightweight edge cloud architectures. In: International Conference on Internet of Things, Big Data and Security (2018)
74. von Leon, D., Miori, L., Sanin, J., El Ioini, N., Helmer, S., Pahl, C.: A Lightweight Container Middleware for Edge Cloud Architectures. In: Fog and Edge Computing: Principles and Paradigms, pp. 145–170 (2019)
75. Warden, P.: Tensorflow 1.9 officially supports the raspberry Pi (2020). https://medium.com/tensorflow/tensorflow-1-9-officially-supports-the-raspberry-pi-b91669b0aa0. Accessed 04 June 2020
76. Xi, B., Xia, C.H., Liu, Z., Zhang, L., Raghavachari, M.: A smart hill-climbing algorithm for application server configuration. In: 13th International Conference on WWW (2004)
77. Yigitbasi, N., Willke, T.L., Liao, G., Epema, D.: Towards machine learning-based auto-tuning of mapreduce. In: IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, pp. 11–20 (2013)