Serverless Container Cluster Management for Lightweight Edge Clouds

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

Keywords: Edge Cloud, Container, Cluster Architecture, Raspberry Pi, Single-board Devices, Docker Swarm, Big Data, Data Streaming, Performance Engineering, Serverless, FaaS, Function-as-a-Service.

Abstract: Clusters consisting of lightweight single-board devices are used in a variety of use cases: from microcontrollers regulating the production process of an assembly line to roadside controllers monitoring and managing traffic. Often, data that is accumulated on the devices has to be sent to remote cloud data centers for processing. However, with hardware capabilities of controllers continuously increasing and the need for better performance and security through local processing, directly processing data on a remote cluster, known as Edge Computing, is a favourable solution. Recent trends such as microservices, containerization and serverless technology provide solutions for interconnecting the nodes and deploying software across the created cluster. This paper aims at proposing a serverless architecture for clustered container applications. The architecture relies on the MQTT protocol for communication, Prometheus for monitoring and Docker swarm in conjunction with openFaas for deploying services across a cluster. Using the proposed architecture as a foundation, a concrete traffic management application is implemented as a proof-of-concept. Results show that the proposed architecture meets performance requirements. However, the network set-up as well as the network capabilities of the used devices were identified as potential bottlenecks.

1 INTRODUCTION

Internet-of-Things (IoT) and edge computing platforms allow to transfer computation and storage away from classical data centre clouds. Communication infrastructures such as the 5G mobile standard aim at enabling communication between edge and IoT devices almost in real time with transfer rates of up to 20 Gbit/s (5G-CARMEN, 2019), driven by low latency needs. In addition to higher transfer rates, they also offers new technologies such as Network Function Virtualization (NFV) allowing the execution of code functions on generic hardware nodes without having to install specific hardware (Kiss et al., 2018). Software Defined Networking (SDN) allows third parties to directly use hardware resources by defining the desired set-up in a programmatic way.

These technologies can be based on the so-called 'serverless' concept. Serverless aims at shifting the responsibility of deploying, scaling and maintaining the software to an infrastructure provider. Instead of deploying and running an application on designated hardware themselves, developers only need to hand it over to the serverless platform. Despite being only a recent trend, serverless technology is already used in a wide variety of cases. Baldini et al. show that interest has been increasing since 2015 (Baldini et al., 2017). So far, published research has focused on trends and shortcomings (Baldini et al., 2017) or reviewing different frameworks (Kritikos and Skrzypek, 2018). In order to provide new experimental evidence, we implement and evaluate a serverless use case application on a lightweight edge cluster architecture. Processing tasks are computed directly on nodes at the edge of a cluster and are not send to remote processing hubs. We evaluate whether a serverless system is a suitable basis for demanding use cases in edge environments. Since small, single-board devices, like the Raspberry Pi, are widely adopted and will potentially keep increasing their processing power, we investigate if and under what conditions a cluster of such devices is able to support a complex, low-latency system that is tightly constrained by a fixed set of requirements. Our proposed architecture is based on industry-standard technologies such as MQTT for inter-cluster communication, openFaas and Docker Swarm for the implementation of the serverless concepts and Prometheus for monitoring. The evaluation validates how the functional and non-functional requirements are addressed in the proposed system. We also analyse the advantages and disadvantages the serverless concept offers over traditional approaches of deployment. Ad-

Gand, F., Fronza, I., El Ioini, N., Barzegar, H. and Pahl, C.
Serverless Container Cluster Management for Lightweight Edge Clouds.
DOI: 10.5220/0009379503020311
In *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*, pages 302-311
ISBN: 978-989-758-424-4

ditionally, we report on the performance of the system for different scenarios. Potential bottlenecks will be identified and discussed.

The paper is structured as follows. In Section 2, concepts and technologies are introduced. Section 3 discusses related work on distributed edge systems. Section 4 describes the high-level architecture before describing each component in greater detail. Section 5 evaluates the architecture. We conclude with a final overview and suggestions for future research.

2 TECHNOLOGY SELECTION

This section introduces key concepts as well as specific tools and technologies. The final subsection describes how the tools and technologies can be combined to create a lighweight edge architecture.

2.1 Background Technologies

Edge Computing is different from cloud computing. Cloud computing is based on data centers that are able to process large amounts of data (Kiss et al., 2018). Data from local systems is usually sent to and processed by the cloud. After the computation is complete, a result may be returned to the local network. This approach, however, leaves the potential local processing power of the network unused and comes with a significantly increased latency. Edge computing leverages the processing power of nodes "at the edge" of the network. These nodes are an intermediate layer between the devices (Kiss et al., 2018).

Microservices have become popular in recent years. Traditional architectures usually ship an application as a monolith: the entire application is bundled into one executable that was deployed on specific hardware. When switching to a microservice architecture, the monolith is split into microservices, running in an independent process and having their own deployment artifacts (Jamshidi et al., 2018).

Serverless Computing is a new concept for the deployment of cloud applications (Baldini et al., 2017). Serverless allows developers to focus solely on the application code without being concerned about the servers it is deployed on. The tasks of managing and scaling the application are handled by the cloud provider: with serverless, the developer can expect the code to be fault-tolerant and auto-scaling. In addition to the major cloud providers already offering serverless functionality, several openSource 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. This work will focus on open source, self-hosted solutions. Serverless is different from PaaS (Platform-as-a-service) and SaaS (Software-as-a-Service) where the deployment of code is specifically tailored to the platform.

- *FaaS*: usually, serverless computing is accompanied by a concept called Functions-as-a-Service (FaaS). Here, small chunks of functionality are deployed in the cloud and scaled dynamically by the cloud provider (Kritikos and Skrzypek, 2018). These functions are usually even smaller than microservices. They are short-lived and have clear input and output parameters, similarly to functions used in most programming languages.
- *Serverless Microservices*: 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 could be considered (Ellis, 2018). Managing and deploying these microservices is similar to serverless functions.

Open-source serverless frameworks that we initially considered are summarised in Table 1. Completeness, licensing model and support for Docker and Prometheus led us to select openFaas.

2.2 Platform Technology Review

The concrete hardware, software and standards used in our architecture shall now be introduced.

The **Raspberry Pi** is a single-board computer based on an ARM-processor. Since the start in 2012 there have been four major iterations of the Raspberry Pi platform. The version 2 B models we use include a 900MHz quad-core ARM Cortex-A7 CPU and 1GB of RAM.

Docker is a containerization software. Containerization is a virtualization technology that, instead of virtualizing hardware, separates processes from each other by utilizing features of the Linux kernel. Docker containers bundle an application along with all of its dependencies. Docker offers the ability to create, build and ship containers. Compared to virtual machines, containers offer a better use of the hosts resource 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 are built using Dockerfiles, which describe the system to be constructed. Docker (the Docker Engine) is based on a client-server architecture. The client communicates with the Docker deamon via a command-line-interface. The docker daemon is in charge of managing the components and

Table 1: Comparison of open-source serverless frameworks.

Framework	Languages	Type	License	Vendor	Monitoring	Key components
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	Python, NodeJS, Ruby, PHP, Go, .NET, Java, Ballerina	Complete Framework	Apache License 2.0	community driven	Prometheus	Kubernetes (native)
Serverless (community) (Serverless, 2019)	Python, NodeJS, Java, Go, Scala, C#	CLI for building + deploying	MIT	Serverless, Inc.	-	Deployment to providers: AWS, MS Azure etc.

containers. Docker services represent the actual application logic of a container in production. Using services, a 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 used to update the site.

Docker Swarm is the cluster management tool integrated into Docker. 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, scaling can also be handled by the swarm.

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. **Ansible** is a tool for automating tasks in a cluster and cloud environments such as individual node configuration or application deployment. **MQTT** is a network protocol designed for IoT. It is primarily used for unreliable networks with limited bandwidth. **Prometheus** is a monitoring tool to gather and process application metrics. 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 is a Functions-as-a-Service (FaaS) framework. It can be deployed on top of a Docker swarm or a Kubernetes cluster. When starting the openFaas framework, some standard docker containers are deployed: Gateway is used as the central gateway for calling functions from anywhere in the cluster. A Prometheus instance is running on this container. The Alertmanager reads Prometheus metrics and issues alerts to the gateway. By default, openFaas contains a simple autoscaling rule that leverages the default metrics aggregated by Prometheus and scales based on predefined thresholds (openFaaS, 2019).

2.3 Requirements & Platform Selection

Among the serverless frameworks compared in Table 1, openFaas was selected for the implementation of the application. The reasons are: (i) a wide array of supported languages, (ii) openFaas being a complete, all-in-one framework, (iii) Prometheus as an integrated, extendable monitoring solution, (iv) a simple set-up process and (v) 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).

To meet the requirements for a lightweight edge platform, the tools and technologies we used need to provide a high level of flexibility while preserving the limited hardware resources of the cluster. By splitting the application into microservices and containerizing it, the hardware can be reallocated dynamically (Mendonca et al., 2019). This also enables scaling the different parts of the application in a simple way. The Docker images leave a minimal footprint on the devices, making efficient use of the hardware. MQTT, as a lightweight protocol, has a similar advantage, while its underlying publisher/subscriber pattern simplifies communication in an environment where services are added and removed constantly. Establishing peer-to-peer communication would be significantly more complex. OpenFaas is a simple way of building and deploying services/functions across the cluster. In

addition to the obvious functionality, some of its features, such as the built-in Prometheus instance or the gateway, can be extended upon to make openFaas the central building block of the application. Even though the openFaas scaling functionality is limited by default, it can be used as a foundation to implement a more fine-grained scaling algorithm using the built-in monitoring options. An overview of the proposed architecture is shown in Figure 1.

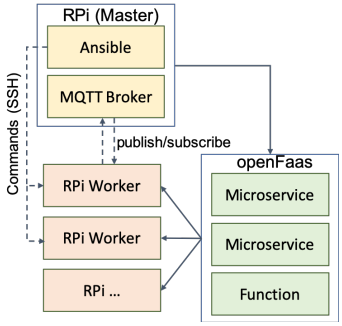

Figure 1: Hardware and networking components.

3 RELATED WORK

The discussion of serverless technology is still in its early stages (Baldini et al., 2017). The authors introduce several commercial implementations such as Amazon's AWS Lambda or Microsoft Azure Functions. Major advantages mentioned are the responsibility of managing servers and infrastructure that no longer rests with the developer.

(Kritikos and Skrzypek, 2018) review serverless frameworks. They highlight the need for new methods to create an architecture for applications that contain both servlerless components like functions as well as classic components such as microservices running inside a Docker Container. 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.

(Kiss et al., 2018) gather requirements for applications using edge computing, specifically combining them with the 5G standard. They mention that recently released single-board devices open up the possibility of processing some of the data at the edge. This, however, creates the challenge of orchestrating the available processing power. The IoT system needs to reorganize itself based on changing conditions.

(Tata et al., 2017) outline the state of the art as well as the challenges in modeling the application architecture of IoT edge applications. A sample scenario 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.

The papers reviewed above 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 prototypes for analysis and experimentation. The work presented here 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 devised solution will be evaluated in a second step.

A system that is comparable to the one proposed here has been presented in (Steffenel et al., 2019). The authors introduce a containerized cluster setup on single-board devices that is tailored towards applications that process and compute large amounts of data. They note that the performance of the RPI cluster is still acceptable and could be a suitable option for comparable use cases. During the evaluation they also identified the poor networking performance of the Raspberry Pis as an additional bottleneck. Additional topics that we will additionally address here, such as a serverless architecture for such systems or scaling options, are not within the scope of that paper.

4 PROPOSED ARCHITECTURE

We now provide an overview of the proposed architecture. The aim is to develop a container management platform, deployed on a cluster of single-board devices, based on the concepts of serverless computing and microservices architecture. We will first introduce the foundation of the application: the underlying hardware and the set-up process of the cluster. In a second step, the building blocks of the application are introduced at a conceptual level before covering low-level implementation details. The proposed binding blocks of the application such as the serverless and microservices-based architecture can be reused for different applications in different contexts. The generic scaling component is applicable in different applications by adjusting a few constants.

4.1 Architecture Overview

The application is composed of three main architecture layers (Pahl et al., 2018). The platform layer represents the hardware architecture of the cluster and the application. The system layer includes the components of the system. On top of these two, the controller layer scales the components of the platform.

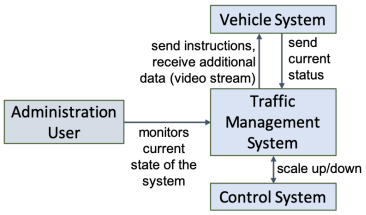

Figure 2: System context: interaction between systems.

Figure 2 shows the interaction between the system layer, the controller layer and additional components.

We apply the generic architecture here to a traffic management application. A Traffic Management System (TMS) manages and coordinates a number of cars in a road section. Services could autonomous driving functions such as maneouvering or state share features. The Vehicle System (VS) represents the cars here. This application is an example of a mobile, low=latency platform that we modelled after the 5G-CARMEN project (5G-CARMEN, 2019). The TMS contains all core functions of the system. There is a constant exchange of messages between the Traffic Management System TMS and the VS that contains simulations of vehicles. The Control System (CS) is used to scale parts of the TMS based on the current situation and a number of predefined factors.

4.2 Platform Architecture

The high-level platform layer, shown in Figure 3, aims at hiding some of the lower layer cluster management to the application. 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. All nodes establish a connection to the master by addressing it by its hostname. Using Docker Swarm and openFaas, the RPIs are connected in such a way 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 the specific node as this abstraction layer is hidden behind openFaas. The services and functions are built and deployed using the openFaas command line interface. Almost all services run the *python:3.6-alpine* docker image. This image is based on Alpine, a minimalistic, lightweight Linux distribution, that is shipped with a python 3 installation. OpenFaas is also used to scale the services independently. Communication between the services is achieved by relying on the openFaas gateway as well as on the MQTT broker.

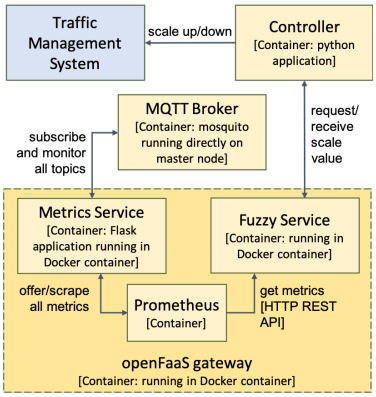

Figure 3: Platform and Application Layer: platform and traffic management application components.

The cluster in our experimental setting 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) (Scolati et al., 2019). The system components are split into three repositories. The rpicluster repository contains the clout-init configuration files for setting up the Raspberry Pis. The rpicluster-application repository ncludes all microservices and scripts that make up the application logic. The last repository includes a modified version of the the openFaas repository. We provide detailled information here in order to give evidence for the feasibility of the setup and provide repeatable configuration and installation instructions.

All nodes of the cluster run HypriotOS. 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.

4.3 Monitoring

Monitoring utilises the openFaas Prometheus instance. 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 collect from. The Prometheus python API is used to implement the metrics collection. The metrics service currently exposes the number of messages, such

as the number of active cars for the TMS/VS as well as the cumulative memory and CPU usage. The number of messages is implemented as a counter which is continuously increasing.

The *Prometheus 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 (**?**). The aggregated data is returned in 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 collects 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 work. 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 without 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.

4.4 Application Feature Requirements

In the following section, the application-level features and non-functional requirements of the traffic management use case are described. This will be done by highlighting which parts of the system address a specific requirement.

- Simulate vehicles: The vehicles are simulated by the vehicle service VS. Communication between the gatherer and vehicle services is implemented using MQTT. Video streams for in-car consumption are received by the central openFass gateway.
- Collect vehicle information: The gatherer service collects and aggregates vehicle information.
- Continuous modeling of current state of the road section: The gatherer service includes an internal representation of the road section.
- Issue commands to vehicles: Commands are issued via MQTT by the gatherer service. Decision-making functionality is both available as a separate serverless function and as built-in functionality within the gatherer service.
- Provide video streaming: The video service instances act as broadcasters. Vehicles access these broadcasters using the central openFaas gateway.
- Provide bus for communication between services: The use of the openFaas gateway and the mosquitto broker offer a way to access services without having to know the address of the hardware node they are running on, thus enabling the simple transparent discovery of new services and communication between components.

The application is split into different services that can be easily deployed across the cluster from a central node. Physical access to individual nodes is not necessary. The system can be reconfigured by using the corresponding central service. Configuration changes are automatically distributed across the nodes by using the MQTT subscriber/publisher pattern. Several gatherer services are usually deployed at the same time. If a gatherer is added or removed the vehicles are automatically distributed among the available gatherers. The scaling functionality ensures that the performance remains above a certain SLO.

5 EVALUATION

The proposed system shall be evaluated in terms of performance and dependability (scalability) concerns.

5.1 Evaluation Objectives

The focus lies on evaluating the performance of the proposed serverless microservice solution.

- Performance: The objectives of the evaluation are as follows. One main goal was to find structural and architectural weaknesses, refactor the system based on the findings and evaluate the effectiveness of the refactoring process. Those weaknesses can be found focusing on parts of the application that introduce performance bottlenecks. The bottlenecks are identified by measuring certain performance metrics that are introduced later on.

  We present the evaluation results of an initial and a refactored system architecture in order to demonstrate the importance of the architectural design in the platform implementation. It will also highlight generic challenges of lightweight clustering and how these can be addressed. Therefore, a two-stage evaluation approach with initial and refactored architectures is essential.
- Scalability: Finally, based on the final, complete set-up of the system, there was a need to evaluate

the maximum number of vehicles the set-up (including the network it was operated in) could support in order to maintain dependability through determining the scalability limits.

These objectives were evaluated for two cluster set-ups. To obtain a first understanding of the system and the possible range of variables, first, an evaluation of a *calibration pilot* was conducted on a small cluster of 3 RPIs. Afterwards, the evaluation procedure was repeated for a *complete cluster* of 8 devices.

5.2 Evaluation Set-up

Performance: 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 autonomous driving functionality. Included in the MRT is the (openFaas) *Function Invocation Time (FIT)* that is listed separately to be able to individually report on the serverless performance. In this evaluation, all MRT and FIT values are considered average values aggregated over the last 20 seconds after the previous scaling operation was completed. In our study, the maximum scale value was unknown. In some real-life scenarios this value might have been specified beforehand. Over the course of this evaluation, different MRT thresholds are applied.

Workload: For all configurations 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. This is possible, because the entire cluster can be seen as one system by combining the individual nodes using Docker swarm and openFaas.

5.3 Evaluation – Default Architecture

The first evaluation round looked at the default architecture with two set-ups (calibration and full).

Round 1 – Calibration Pilot: The evaluation was started with a calibration cluster consisting of three RPIs: a master and two worker nodes. The maximum scale value was set to 5 in order to avoid scaling the system to a point the set-up could not handle anymore.

Table 2 reports on the initial metrics for different numbers of vehicles. The scaling functionality was enabled during monitoring, with its variables set to the values reported in Figure 3. Table 3 includes the data of the scaling algorithm for an initial run (Gand et al., 2020). The number of scaled components is also reported. Manually set variables were Maximum Scale Value: 5, MRT Threshold: 2.0 seconds.

Table 2: Results for a cluster of three RPIs.

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: Scaling run for a cluster of three RPIs with a (fixed) number of 12 cars – with IT: Invocation Time after scaling, Df: Decision Function.

Iteration	IT	# gatherer	# DF
1	1.74	10	5
2	2.5	10	5
3	2.05	10	5
4	1.9	12	6

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 for 14 vehicles, the MRT is already above 10 seconds, which is a value that is too high for many real-world scenarios. With only three RPIs and the given default setup, the performance of the system is limited.

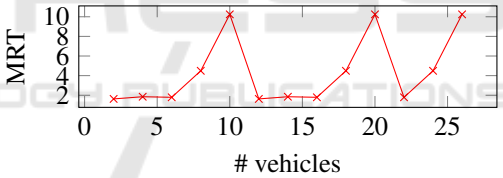

Figure 4: Average MRT - 8 RPI cluster, increase of vehicles.

Round 1 – Full System: Based on the initial findings, the complete cluster of eight RPIs was evaluated and the data was accumulated in Table 4. The cluster consisted of one master node and seven worker nodes. CPU usage started out at about 30% and only showed a slow increase as the number of cars within the system was rising. Therefore, CPU/memory usage did not seem to be the limiting factor. However, the invocation time as well as the MRT were again surprisingly high with the Function Invocation Time starting out at 1.5 seconds and increasing to a value of over 2 seconds at only eight vehicles. The MRT values are also fluctuating greatly as can be seen in Figure 4. Looking at these values, the bottleneck appears to be the openFaas Function Invocation Time. Figure 5 shows that the Function Invocation Time takes up a significant proportion of the overall MRT.

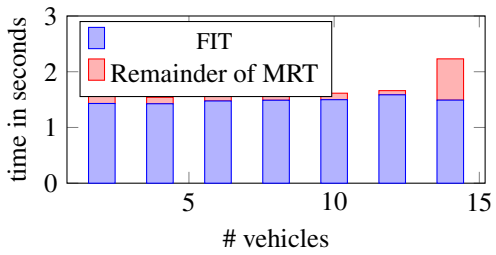

Figure 5: FIT for overall MRT - different no. of vehicles.

5.4 Performance Re-engineering

With the Function Invocation Time (FIT) as a significant factor in slowing down the system, changes to the architecture of the application needed to be made in order to improve the overall performance.

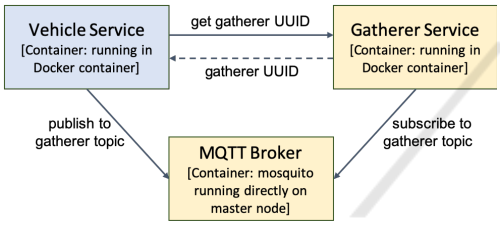

Figure 6: Refactored Architecture: communication between gatherer and vehicle services in the improved system.

The use of parallelism was increased by deploying more than one gatherer. Previously, a single gatherer was processing incoming messages in an asynchronous way and calling the decision-function for each message. This was a bottleneck since the number of messages that could be processed concurrently was limited by the hardware. In the new version, the gatherer itself is subject to scaling. An arbitrary number of gatherers can be deployed across the cluster. Since the vehicles now need to be distributed across the different gatherers, a new component was implemented that computes a unique ID for each gatherer at startup and exposes it via an openFaas function. When a new vehicle is entering the system, it calls the corresponding function once and is assigned a gatherer. Consequently, the vehicle publishes its information to the MQTT topic of its designated gatherer. OpenFaas alternates between the gatherers for each new call by default. Figure 6 highlights the described changes by emphasizing the communication between the vehicle and gatherer services. When new gatherers are added or removed, a message is broadcasted and all vehicles are being reassigned new gathers. Additionally, since the FIT of the decision-function seemed to be a major reason for the slow overall response time, the decision-function was removed and its functionality was included as part of the gatherer. Using this setup, it is no longer necessary to call an openFass function for each received message and wait for its return value. This demonstrates the value of experimental platform evaluation and subsequent refactoring. The aim here is to employ the concept of *continuous experimentation* in order to continuously improve metrics, such as performance in this case.

Table 4: Results for the standard set of metrics for a cluster of eight RPIs using the original version of the system.

Vehicles	Memory	CPU	MRT	FIT
2	37.06	34.72	1.6	1.43
4	22.24	33.0	1.54	1.43
6	19.6	33.28	1.56	1.48
8	29.9	34.32	1.59	1.49
10	29.92	35.37	1.61	1.5
12	34.82	36.17	1.59	1.51
14	35.94	37.27	2.23	1.49
16	42.59	38.57	1.72	1.55
18	49.26	39.42	1.68	1.63
20	51.73	39.92	4.17	1.63
22	52.97	40.53	2.20	1.73
24	66.97	42.37	5.41	2.13
26	59.74	43.28	26.62	1.91

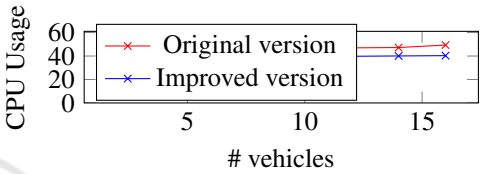

Figure 7: CPU Usage compared between the original and improved (no openFaas function calls) versions.

5.5 Evaluation – Refactored Architecture and Scalability

The experimental setup was similar to the previous full system evaluation: a cluster of eight RPIs was used. As discussed, the decision-making functionality is included in the gatherer service, which can now be scaled independently. Hence, there is no longer the need to call the decision function for each message.

The results can be found in Tables 5 and 6. Considering these results, a significant improvement in the overall Message Roundtrip Time can be noted. The difference between the two versions can be clearly seen by looking at Figure 8 and comparing the MRT of both iterations of the system. This difference was recorded for different numbers of vehicles. It seems to be growing exponentially. Between 16 and 24 vehicles, the MRT increases by over 300% for the original version, while the MRT of the bundled version only increases by about 8%. If we compare the CPU usage, we see that the refactored system uses

Table 5: Results for the standard set of metrics for a cluster of eight RPIs using the improved system. The FIT was not measured since the decision function was removed.

Vehicles	Mem Usage	CPU Usage	MRT	FIT
2	2.92	36.53	0.02	-
4	3.29	37.02	0.025	-
6	3.89	37.90	0.029	-
8	5.48	38.41	0.028	-
10	5.81	39.37	0.17	-
12	5.17	39.78	0.028	-
14	6.98	40.07	0.028	-
16	4.2	40.33	0.025	-
18	4.5	40.67	0.025	-
20	4.94	41.1	0.027	-
22	5.19	41.43	0.028	-
24	5.74	41.7	0.027	-
26	5.99	42.14	0.028	-

Table 6: Same as in Table 5, here higher number of vehicles.

Vehicles	Mem Usage	CPU Usage	MRT	FIT
50	8.63	46.75	0.03	-
75	11.79	51.7	0.032	-
100	14.48	56.57	0.04	-

about 8-10% more of the CPU compared to the original application. The additional computing power and time needed to make a decision is neglectable when compared to the significant overall MRT advantage.

The current set-up did not allow for more than 75 vehicles. The bottleneck appeared to be the network: when trying to increase the number of vehicles beyond this, the network was unable to handle the message volume that were exchanged, which resulted in connections and packets being dropped continuously.

5.6 Overall Evaluation Analysis

Using a smaller cluster set-up (the calibration pilot), it was possible to derive starting values for all variables. The overall evaluation shows that containerization comes with a small performance loss compared to traditional set-ups. However, the advantages of using our approach are generally more significant than

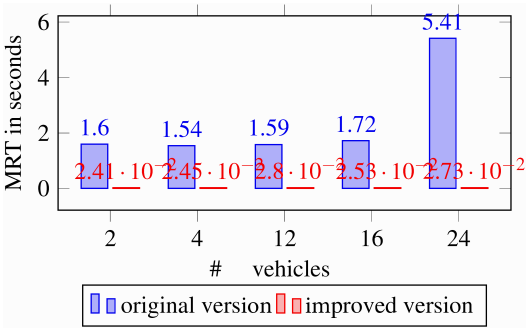

Figure 8: Comparing MRT - original and improved system.

the downsides. The evaluation indicates that serverless function calls should be managed carefully since they introduce network latency problems. Refactoring the proposed solution to reduce the number of necessary calls to openFaas functions resulted in a significant increase in performance. The improved version yields satisfying results in terms of hardware consumption and performance (MRT) while the built-in scaling algorithm scales the system as intended. The network emerges as the limiting factor. Our set-up was not able to process more than 75 vehicles 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 should be able process more. Future improvement attempts could find network set-ups that allow dependable services beyond the limits.

6 CONCLUSIONS

We introduced a containerized serverless edge cluster management, implemented for a use case for a traffic management system on a cluster of single-board devices. The presented architecture is based on edge computing principles, clusters of single-board devices, microservices, serverless technology and auto-scaling. The proposed architecture 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. While we have used a traffic management system for implementation, the architecture itself is generic. Vehicles in the implementation were only simulated, representing actors that continuously produce data into the system.

The advantages of our solution are reusability, scalability and interoperability. By using openFaas beyond its documented boundaries, it was possible to utilise the framework for inter-service communication as well as for monitoring. The bottleneck that prevents the system from scaling even higher appears to be the network infrastructure as well as the limited internal networking capabilities of the RPi. More research in terms of network configuration and management would be here beneficial.

Future work could focus on a number of aspects in addition to the network concern already addressed, e.g., improving traffic management components to drive them towards a more realistic behavior.

REFERENCES

5G-CARMEN (2019). 5G-CARMEN - 5G for Connected and Automated Road Mobility in the European UnioN. https://www.5gcarmen.eu/.

Azimi, S., Pahl, C. and Shirvani, M. H. (2020). Particle swarm optimization for managing performance in multi-cluster IoT edge architectures. In *Intl Conf 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.

Ellis, A. (2018). Introducing stateless microservices for openfaas. https://www.openfaas.com/blog/. Accessed: 2019-11-11.

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.

Gand, F., Fronza, I., Ioini, N. E., Barzegar, H. R., Azimi, S., and Pahl, C. (2020). A fuzzy controller for self-adaptive lightweight container orchestration. In *Intl Conf on Cloud Computing and Services Science*.

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).

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., 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. *QoSA*, 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.

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.

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 International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 161–168.

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*.

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*.

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

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. (2018). Architectural principles for cloud software. *ACM Transactions on Internet Technology (TOIT)*, 18(2):17.

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. (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*.

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 *9th International Conference on Cloud Computing and Services Science*.

Serverless (2019). Serverless framework. https://serverless.com/. Accessed: 2019-11-13.

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'19*.

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*.

Tata, S., Jain, R., Ludwig, H., and Gopisetty, S. (2017). Living in the cloud or on the edge: Opportunities and challenges of iot application architecture. In *2017 IEEE Intl Conf on Services Computing (SCC)*.

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.