Distributed On-Demand Deployment for Transparent Access to 5G Edge Computing Services

Josef Hammer, Hermann Hellwagner
Institute of Information Technology, Alpen-Adria-Universität Klagenfurt, Austria
{ josef.hammer, hermann.hellwagner }@aau.at

***Abstract*—Multi-access Edge Computing (MEC) is a central piece of 5G telecommunication systems and is essential to satisfy the challenging low-latency demands of future applications. MEC provides a cloud computing platform at the edge of the radio access network. Our previous publications argue that edge computing should be transparent to clients, leveraging Software-Defined Networking (SDN). While we introduced a solution to implement such a transparent approach, one question remained: How to handle user requests to a service that is not yet running in a nearby edge cluster? One advantage of the transparent edge is that one could process the initial request in the cloud. However, this paper argues that on-demand deployment might be fast enough for many services, even for the first request. We present an SDN controller that automatically deploys an application container in a nearby edge cluster if no instance is running yet. In the meantime, the user's request is forwarded to another (nearby) edge cluster or kept waiting to be forwarded immediately to the newly instantiated instance. Our performance evaluations on a real edge/fog testbed show that the waiting time for the initial request – e.g., for an *nginx*-based service – can be as low as 0.5 seconds – satisfactory for many applications.**

***Index Terms*—Multi-Access Edge Computing, MEC, Fog Computing, Software-Defined Networking, SDN, Serverless Computing, Container, Docker, Kubernetes**

I. Introduction

The ever-increasing processing and storage demand of the next generation of Internet of Things (IoT) applications led to an evolution in distributed computing and initiated the transition of cloud services closer to the end users [1]. One promising technology is *Multi-access Edge Computing (MEC)* [2]. MEC services and convenient access to them are important building blocks of 5G networks and applications. Particularly IoT applications can benefit significantly from low-latency and high-bandwidth offloading capabilities. Benefits may include lower battery consumption and access to the latest algorithms without requiring to update the software on the device itself.

Therefore, in [3] and [4], we proposed an efficient solution to *transparently* redirect requests for cloud services to the corresponding services running in local edge hosts (see section II and fig. 1). This solution, based on Software-Defined Networking (SDN) capabilities, (i) simplifies the development of applications that use edge computing and (ii) allows existing applications to use edge computing without any modifications.

Unfortunately, such a transparent approach can only work well if the edge services are already running in an edge cluster nearby. With potentially millions of edge services, this seems highly unlikely. Of course, prediction algorithms could be used to pre-deploy the required services just in time. However, a hundred percent correct prediction rate is impossible, even if machine learning is used to predict future requests (e.g., [5]). Thus, there will always be plenty of requests that the SDN controller cannot redirect to a nearby edge cluster but must forward to the cloud instead. The latter, however, may not be suitable for applications that require locality awareness or are very bandwidth-intensive.

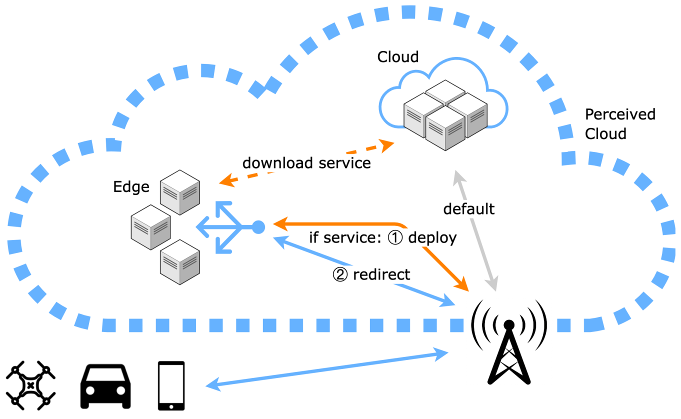

Fig. 1. Perceived cloud vs. real cloud. All requests/responses look like cloud accesses to the client (user equipment; UE) – the redirection to the edge is transparent. For requests to registered services, the service is deployed either proactively or on demand on the first request. Depending on the scheduler's decision, the first request is sent to the newly deployed instance or another instance already running in a different edge cluster.

Thus, in this paper, we address this crucial piece of a transparent solution and focus on a simple question: Is it feasible to keep a client's request to a non-running edge service on hold while deploying the *containerized* service on demand in a nearby edge cluster? While alternative virtualization technologies like *WebAssembly* [6] are faster to launch [7], our focus is on containers as they offer more flexibility regarding the type and complexity of applications. We compare the response times for clients' requests when deploying two containerized edge services on-demand on Kubernetes – the most popular container management system [8] – and Docker – for a lightweight alternative. Alternatively, we consider redirecting the request to a nearby edge for low-latency applications while deploying the new service instance. Once the new instance runs, we redirect requests to the newly created service instance.

The main contributions of this work are:

- an SDN controller design and open-source implementation [9] that deploys containerized applications on-demand to Docker and Kubernetes clusters and allows the loading of different scheduler algorithms;
- a three-phases definition of the deployment process of containerized services;
- an automated annotation of service definition files for simplified edge service development; and
- a thorough evaluation of the deployment times of four containerized edge services, representing a variety of edge service types, on two different types of edge clusters.

The paper has eight sections. First, we provide a short overview of the approach to *transparent access* to edge computing services in section II and survey the related work in section III. Then, we present our concept in section IV, followed by the design of our SDN controller in section V. The corresponding evaluation is shown in section VI, with the discussion in section VII. Section VIII concludes the paper.

II. Background: Transparent Access

In our *transparent access* approach [3], [4], all requests/responses look like cloud accesses to the client (user equipment; UE) – the redirection to the edge is transparent (fig. 1). The services to be redirected to the edge are first registered with a mobile edge platform provider, identified by their unique combination of domain name/IP address and port number. The network (i.e., an SDN switch) intercepts any request from a client to a registered service and automatically redirects it to the closest available edge server. The edge service processes the request and responds to the client (user equipment (UE) in 5G terms). To achieve transparency toward the clients, our approach uses the packet filtering and rewriting capabilities of *OpenFlow* [10] (see fig. 2). We refer to [3] for a description of the fundamental solution and to [4] for more details. The latter paper focuses in particular on the performance aspects of our approach. It presents a prototype implementation of our SDN controller based on the *Ryu framework* [11] and an evaluation of our prototype on a real edge testbed.

III. Related Work

This section reviews existing approaches for transparent access and the deployment and scheduling of edge services.

Regarding transparent access, Taleb et al. [12] implement their proposed *Follow-me Cloud* using OpenFlow. In [13] they adapted their system to a *Follow Me Edge-Cloud* that works in the proposed ETSI MEC environment [14]. The goal of the underlying concept is to provide a continuous connection even in case either the user or the service migrates to a different location. Schiller et al. developed CDS-MEC [15], an NFV/SDN-based MEC platform, which provides both edge application provisioning and traffic management. It also relies on OpenFlow to redirect client requests.

Regarding deployment and scheduling, Fahs et al. [16]–[18] propose a proximity-aware traffic routing and scheduling system that integrates with Kubernetes. Their solution routes

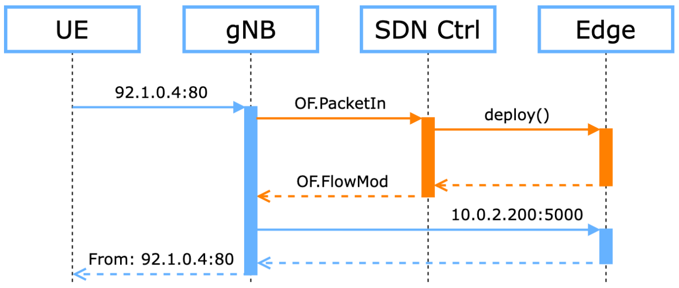

Fig. 2. Routing with a registered service address (IP + port): Using OpenFlow (OF), the request is redirected to the closest edge server, transparent to the client (UE). For subsequent requests to the same service, the redirection rule is already known to the switch (gNB); the packet is forwarded directly to the edge host (and not to the SDN controller anymore). If no service instance is already running in the chosen edge, the SDN controller first deploys and launches a new one before redirecting the client's request to the new instance.

to the optimal Pod using *iptables*. It can only start from a Kubernetes node, while our solution is independent of the cluster type and picks up the request already at the network's ingress – using the more efficient network layer. Quang et al. [19] propose a scheme that enables device-driven on-demand deployment of *serverless computing functions* to the edge. Their work focuses on IoT frameworks, specifically on AWS Greengrass [20]. The IoT devices communicate via MQTT with a monitoring function at the edge to trigger a deployment.

Zanni et al. [21] measure deployment times of containers on Raspberry Pis but not the total time until the application is ready. Watanabe et al. [22] propose an architecture for enabling clients to offload parts of applications to MEC servers, with Kubernetes as the container orchestration system. Mohan et al. [23] show that the creation and initialization of network namespaces account for 90 percent of the startup time of a container. Gackstatter et al. [7] propose that serverless applications using alternative virtualization technologies like WebAssembly [6] might be a promising candidate for edge services since the *cold start* latency is much lower than for container-based solutions. Gadepalli et al. [24] and Shillaker et al. [25] propose efficient frameworks based on WebAssembly. However, containers still offer better security and more flexibility regarding the type and complexity of applications.

IV. On-Demand Deployment

This section presents our concept for the on-demand deployment of edge services. As explained in the introduction, the SDN controller redirects incoming requests to edge services to a specific edge cluster running an instance of the required service. If such an instance is available, the client's request is redirected immediately to the chosen instance, following the *transparent access* approach. However, what happens if no instance is running already in the selected edge cluster?

A. On-Demand Deployment Types

1) On-Demand Deployment with Waiting: The first approach is to keep the client's request on hold while a new instance is launched (see fig. 2 and fig. 5). Since no flow

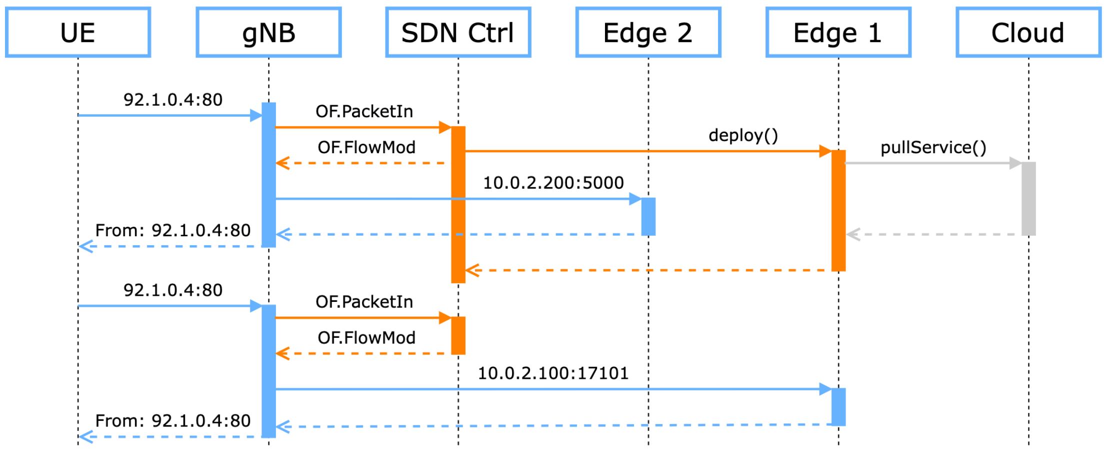

Fig. 3. If the scheduler demands a very low response time, the SDN controller redirects the initial request to a running service instance in an edge further away. In parallel, the controller triggers the deployment of the service in the optimal edge (which includes pulling/downloading the service if not yet cached). As soon as the new instance is running, requests are redirected to this optimal location.

is known to the switch yet, the latter forwards the client's request to the SDN controller. Then, as no service instance runs in the selected edge cluster, the SDN controller triggers a deployment. Ideally, the required service image is cached already on the chosen edge cluster. Otherwise, the edge cluster must pull the required image from the cloud first. Only when the new service instance is up and running, the SDN controller responds to the switch and instructs it to redirect the request to the newly created instance. Finally, the client's request gets forwarded to the newly created service instance, which processes the request and responds to the client.

2) On-Demand Deployment without Waiting: On the other hand, if the scheduler demands a very low response time, the second approach to on-demand deployment is chosen. Here, the SDN controller instructs the switches to redirect the initial request to a running service instance in another edge (possibly further away; see fig. 3) or even to forward it to the cloud. In parallel, the controller triggers the deployment of the service in the optimal edge cluster (which includes pulling/downloading the service from the cloud if not yet cached). Future requests to the same service are redirected to this optimal location as soon as the new instance is running. Keep in mind that edge clusters are usually organized hierarchically. Clusters in close vicinity of the users tend to be smaller, with cluster size and performance growing when further away (i.e., located closer to the "cloud"). As a result, a "non-optimal" (further away, but on the route to the cloud) edge cluster is much more likely to have the requested service cached or even running already.

B. Scheduling and Dispatching

The decision for a specific edge cluster is taken by the `Scheduler` component of our SDN controller. Thus, to keep our system flexible, the concrete scheduler implementation can be defined in the controller's configuration and will be dynamically loaded. Furthermore, our system architecture includes a `Dispatcher` component, which feeds the `Scheduler` with information about the current system state and is responsible for checking and triggering the deployment of edge services. This component also tracks the clients' current location.

We distinguish two types of schedulers (fig. 6):

1) Global Scheduler: responsible for choosing the appropriate edge cluster. It is aware of which edge clusters provide existing and running service instances and returns two results:

a) FAST: The fastest location for the current request.

b) BEST: The best location for future requests.

The latter is returned empty if equal to the `FAST` choice; if non-empty, we have *On-Demand Deployment without Waiting*. If `FAST` is empty, the request is forwarded toward the cloud.

2) Local Scheduler: responsible for choosing a specific service instance within an edge cluster. With a Kubernetes cluster, the K8s scheduler might represent the *Local Scheduler*; however, we might also use a different one. Furthermore, for Kubernetes, we can even define a custom scheduler – like the ones in [26], [27] – to be used for our edge services only.

C. Deployment Phases

In this work, we distinguish three phases for the deployment of an edge service instance (fig. 4). First, unless already cached, the edge cluster needs to `Pull` the required container images or, with *serverless computing*, download the source code from the cloud. Second, the service needs to be `Created`. For Docker, we define this as creating a container; for Kubernetes, a new *Deployment* and *Service* are created (with zero replicas). Third, the edge cluster needs to `Scale Up` a new instance. For Docker, the container is started; for Kubernetes, the number of replicas is increased. When the service is not required anymore, either the controller or cluster may `Scale Down` and even `Remove` the service. The latter phase removes the *Docker* containers and *Kubernetes Deployments* and *Services*. Optionally, but unlikely, the cached items may also be `Deleted` if disk space is scarce. Even if a container image is deleted, some of its layers may be used by other images. Therefore, the next time the system pulls the same image again, it may no longer have to pull *all* layers.

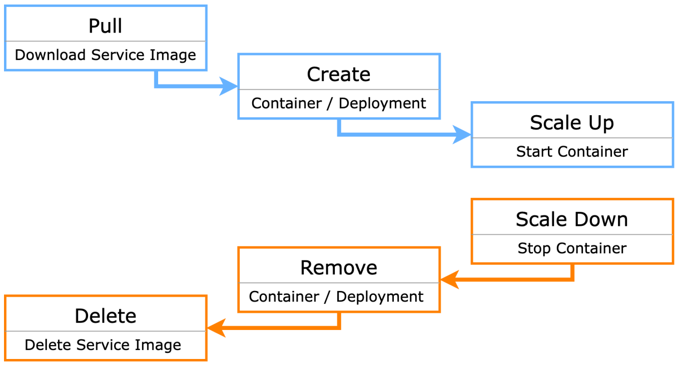

Fig. 4. We distinguish three phases for the deployment of an edge service instance. First, unless already cached, the edge cluster needs to `Pull` the required container images from the cloud. Second, the service needs to be `Create`d. For Docker, we create a container; for Kubernetes, a *Deployment* and *Service* are created (with zero replicas). Third, the edge cluster needs to `Scale Up` a new instance. For Docker, the container is started; for Kubernetes, the number of replicas is increased. When the service is not required anymore, the controller or cluster may `Scale Down` and even `Remove` the service. The latter removes the *Docker* containers and *Kubernetes Deployments* and *Services*. Optionally, even the images may be `Delete`d.

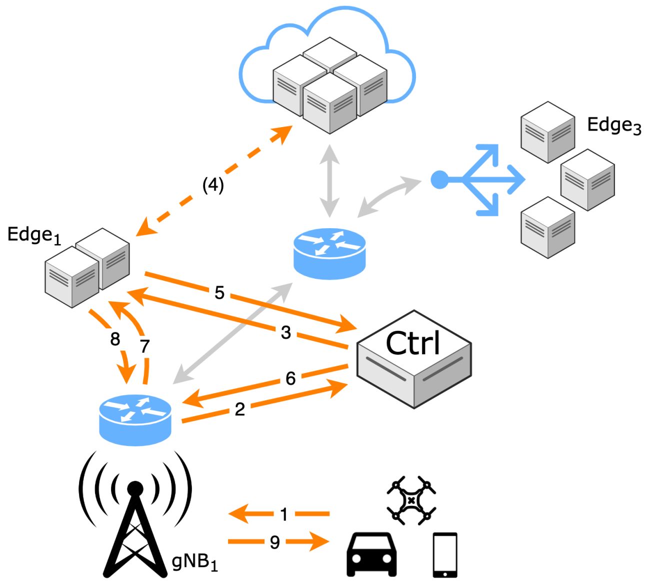

Fig. 5. On-demand deployment *with waiting*: The user's request is kept waiting until a new service instance has been deployed. When a user sends a request (1) to a new service, the switch forwards it to the SDN controller (2). Since no service instance runs in the nearest/optimal edge cluster, the controller triggers a deployment (3). The edge cluster might first have to pull the required service image from the cloud (4). Only when the new service instance is up and running, the controller responds to the switch and instructs it to redirect the request to the newly created instance (6). Finally, the user's request gets sent to (7) and is answered by the newly created instance (8, 9).

V. Design

This section presents the design decisions we made for the on-demand deployment components of our SDN controller. First, we decided to not only instruct the switches with the flows that redirect requests to specific edge service instances. Instead, we also memorize all these flows in a component called `FlowMemory`. This approach allows us to keep the *idle timeout values* in the switches low – if a request from the same client to the same service comes in later, we can redirect the request to the same service used before. However, also the memorized flows have an *idle timeout* after which they are removed from the `FlowMemory`. Apart from removing stale flows, these timeouts serve a second purpose: Our controller may automatically `scale down` idle edge service instances.

First, however, let us look at how the `Dispatcher` and the `Scheduler` decide where to create a service instance. Fig. 7 shows the SDN controller's dispatching algorithm flow chart. If no memorized flow exists in the `FlowMemory`, the `Dispatcher` component gathers a list of existing and running instances of the requested service. Next, the `Dispatcher` passes this list to the `Scheduler` component. The latter returns two choices: a `FAST` choice for the current request, and a `BEST` choice for future requests. Then the `Dispatcher` checks whether the two service instances still need to be created and scaled up. For communicating with Docker and the Kubernetes cluster, we use the respective Python client libraries (see [9]). Once the `FAST` service is available, the controller instructs the switch(es) to redirect the client's request to that service instance.

So, where does the SDN controller get the necessary information about the services? Each edge service needs to be defined in a separate *YAML* file. We use the established and well-defined *Kubernetes Deployment* definition file format. It does not matter whether the edge cluster is running Docker or Kubernetes – we use the same service definition for both. The only mandatory data is the name of the image. In addition to that, the file may include anything valid in a *Kubernetes Deployment* definition. For Docker, only a subset of the values (like volume mounts) are currently parsed.

For deployment in a Kubernetes cluster, our system automatically annotates the service definition with additional information. This approach keeps the system flexible, the definition files lean, and lifts some of the burdens from the developers. First, we automatically set a unique worldwide name for each service – something developers may easily forget when developing their own services only. Second, we add all the `matchLabels` required by Kubernetes. In addition to these, we also add a label named `edge.service` to be able to address and query edge services in the cluster distinctly. By default, we set the number of `replicas` to `0` (*"scale to zero"*). If a *Local Scheduler* (fig. 6) has been defined in the controller configuration for the particular edge cluster, we set it as the value for the `schedulerName` key.

Finally, we automatically create a *Kubernetes Service* definition (unless the developer already included one in the YAML file). In addition to the unique name and the labels, the generated *Service* definition will contain the exposed port where the service can be reached, the target port of the service instance, and, by default, `TCP` as the protocol. Our system also adds labels to Docker deployments to allow addressing and querying edge services distinctly. Volume mappings to the host file system are supported. For details we refer to [9].

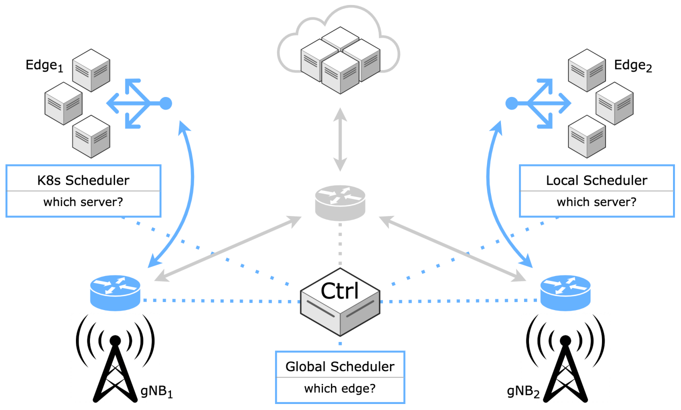

Fig. 6. We distinguish two types of schedulers: A *Global Scheduler* is responsible for choosing the appropriate edge cluster, while a *Local Scheduler* is responsible for choosing a specific service instance within an edge cluster.

VI. Evaluation

For our research on edge/fog computing we deployed a real testbed named *Carinthian Computing Continuum* (C^3) presented in [1]. C^3 aggregates a large set of heterogeneous resources and includes an edge computing layer. The entry point to the edge layer is the *Edge Gateway Server (EGS)*, a 64-bit x86 system with an AMD Ryzen Threadripper 2920X (12 cores, 3.5 GHz, 32 GiB memory) running Ubuntu 18.04. Among the other edge resources are 35 Raspberry Pi 4B running Raspberry PI OS (Buster) – all with four cores and 4 GiB of memory. The EGS supports 10 Gbps Ethernet; the other nodes utilize 1 Gbps Ethernet. A layer-3 HP Aruba switch with 1 Gbps ports connects the devices. For our experiments, we use an overlay network that forms a virtual emulation network. Fig. 8 shows the virtual network topology used for the evaluation. The SDN controller, the virtual OVS switch, the Kubernetes (K8s) cluster, and Docker run on the *Edge Gateway Server (EGS)*; the clients on 20 Raspberry Pis.

We evaluated our on-demand deployment approach using four different edge services (table I). We chose an *Nginx* web server for a service representing a typical use case and container image size. For the previous years, Nginx has been the most popular container image, used by nearly 50 percent of organizations that use containers [8]. Additionally, as an extreme case, we chose a web server written entirely in Assembler, *asmttpd [28]*, representing the smallest and fastest web service possible (image size: just 6.18 KiB!). This webserver's negligible launch time allows us to measure the minimal overhead of starting a service in a container. Both edge services read a short plain-text file and return its content.

Our focus here is on the time required to start a service, and we assume most edge services will be small microservices instead of heavyweight applications. Nevertheless, we also added two additional edge services to evaluate other scenarios. As an example of a slightly more heavyweight service, we decided on an edge service for image classification. Since images and videos require a lot of bandwidth, such services are ideal candidates for edge services. We chose a *TensorFlow*

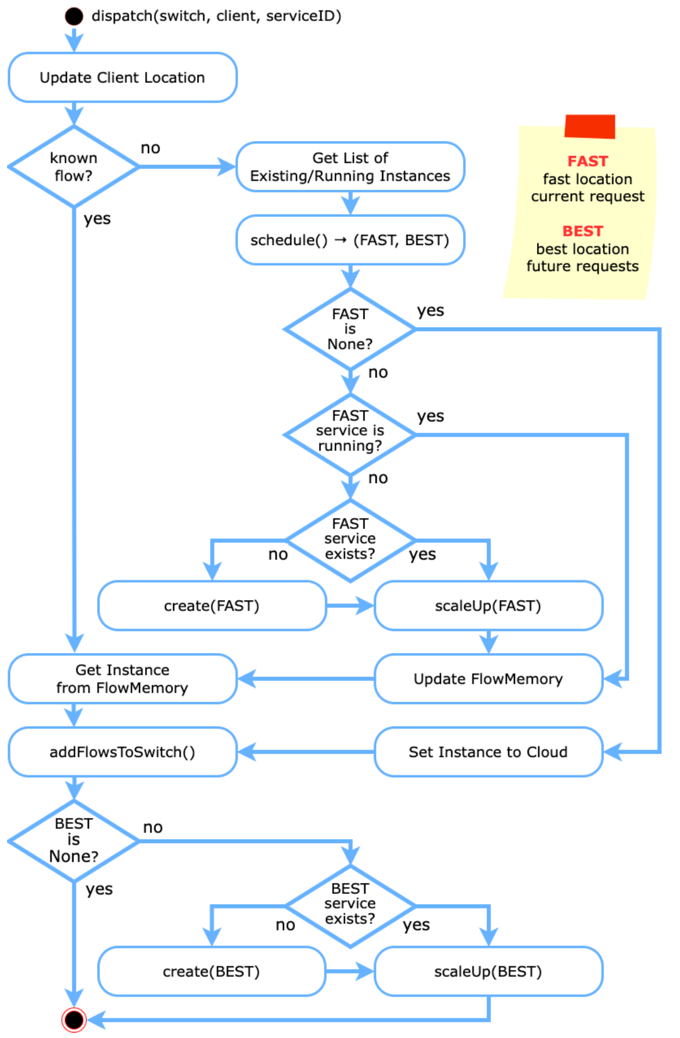

Fig. 7. The SDN controller's dispatching algorithm. If no flow is memorized in the `FlowMemory`, the `Dispatcher` gathers a list of available and running instances and passes it to the `Scheduler`, which returns two choices: A `FAST` choice for the current request, and a `BEST` choice for future requests. If necessary, both chosen instances are created and scaled up.

Serving container with a pre-trained ResNet50 model built into the container. Loading a model takes time; thus, we expect a higher startup time for this service. With this service, the clients send a cat picture for classification (83 KiB payload).

More complex applications usually combine several microservices. Thus, with our fourth edge service, we wanted to test how multiple containers impact deployment performance. We combined the already mentioned *Nginx* with a simple *Python application*. On startup, the application reads environment variables and configuration data from a folder shared by the host system. The gathered information and the current timestamp are then written once per second to the *index.html* file shared with the *Nginx* container.

These four services were evaluated in two different edge cluster environments: a Kubernetes (K8s) cluster and a Docker "cluster", both running on the powerful EGS (see fig. 8). Note that both Kubernetes and Docker use the same *containerd*

TABLE I
EDGE SERVICES USED IN THIS WORK.

	Service	Image(s)	Size / Layers	Containers	HTTP
Asm	Assembler Web Server (asmttpd [28])	josefhammer/web-asm:amd64	6.18 KiB / 1	1	GET
Nginx	Nginx Web Server	nginx:1.23.2	135 MiB / 6	1	GET
ResNet	TensorFlow Serving with pre-trained ResNet50 model	gcr.io/tensorflow-serving/resnet	308 MiB / 9	1	POST
Nginx+Py	Nginx Web Server + Python Application	nginx:1.23.2 + josefhammer/env-writer-py	181 MiB / 7	2	GET

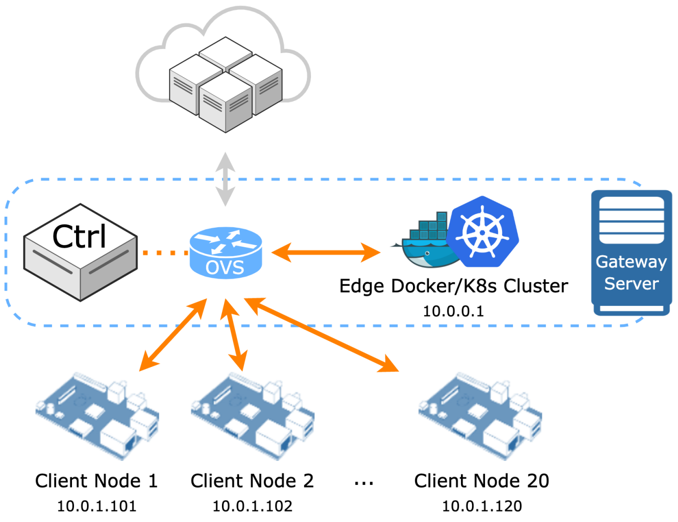

Fig. 8. The topology used for the evaluation. The SDN controller, the virtual OVS switch, Docker, and the Kubernetes (K8s) cluster run on the *Edge Gateway Server (EGS)*. The client applications run on 20 Raspberry Pis.

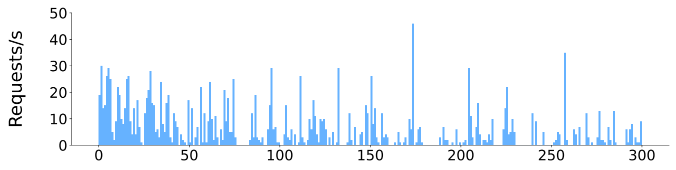

Fig. 9. Distribution of 1708 requests to 42 different edge services over five minutes (from a real network traffic dataset). If a service is not running yet, it will be deployed by the SDN controller – leading to the distribution in fig. 10.

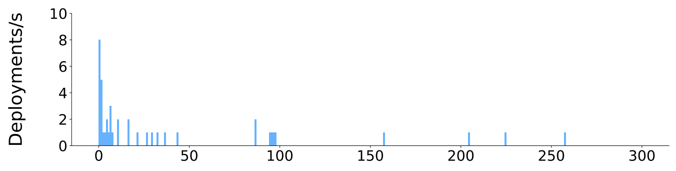

Fig. 10. Distribution of 42 edge service deployments over five minutes – with up to eight deployments per second in the beginning.

container runtime on the EGS. We evaluated all three deployment phases shown in fig. 4. To emulate varying and realistic numbers of requests coming in simultaneously, we utilized the five-minute `bigFlows.pcap` [29] real network traffic capture. We extracted all TCP conversations to public IP addresses and filtered for requests to port 80. As edge service addresses, we selected all destination addresses receiving a minimum of 20 requests – leading us to 42 services receiving 1708 requests (fig. 9). We use a single service type per test run. Every time a service instance is not running yet, it will be deployed by the SDN controller – leading to the deployment distribution shown in fig. 10. Note that the *total times* shown in the following figures (except the pull times in fig. 13) represent the total time for a client requesting a response from an edge service, i.e., from when the client sends the request until it receives the response. We measured the times using our *timecurl.sh* [30] script. The `time_total` provided by Curl includes everything from when Curl starts establishing a TCP connection until it gets a response for the `HTTP` request.

The most critical phase is the `Scale Up` phase. This step cannot be avoided by caching artifacts on a disk – which would be cheap compared to keeping an idle instance running. Fig. 11 shows the total time when the four services only require to be scaled up. The numbers highlight the significant difference between just starting a container via Docker (less than one second) and the overhead of starting the same container on a complex orchestrator like Kubernetes (around three seconds). Note that after scaling up an edge service, our SDN controller needs to wait until the service is fully started and ready to serve (fig. 14 and fig. 15). Therefore, before setting up the flows, the controller continuously tests if the respective port is open. Otherwise, with the port still closed, the server would reject the client's request.

Interestingly, there is no notable difference between starting the tiny Assembler web server and the far larger Nginx instance. As expected, ResNet takes significantly longer to start; the waiting time alone (fig. 14) accounts for more than a fourth of the total time. If the containers for the service do not exist yet, we need to both `Create` and `Scale Up` the service. Fig. 12 shows that creating the containers adds around 100 ms to the response time of the first request – except for ResNet, which shows no overhead.

If the service image is not yet cached on the edge cluster, the next critical phase is the `Pull` phase. Fig. 13 shows the total time to *pull (download)* the different service container images onto the *Edge Gateway Server (EGS)* from *Docker Hub* or – in the case of the ResNet container – from *Google Container Registry*. When pulling the same images from a private container registry located in the same network, pull times improve by about 1.5 to 2 seconds. The `Pull` phase

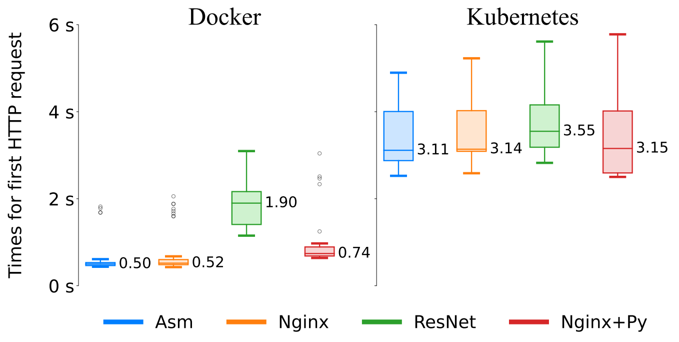

Fig. 11. Total time (median) to *scale up* four different services on two different clusters. We scaled up 42 instances for each test (see fig. 10). The numbers highlight the overhead of an orchestrator like Kubernetes (K8s).

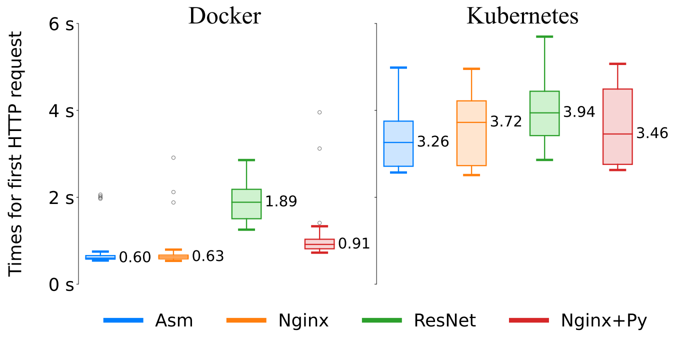

Fig. 12. Total time (median) to *create* + *scale up* four different services on two different clusters. We created + scaled up 42 instances for each test.

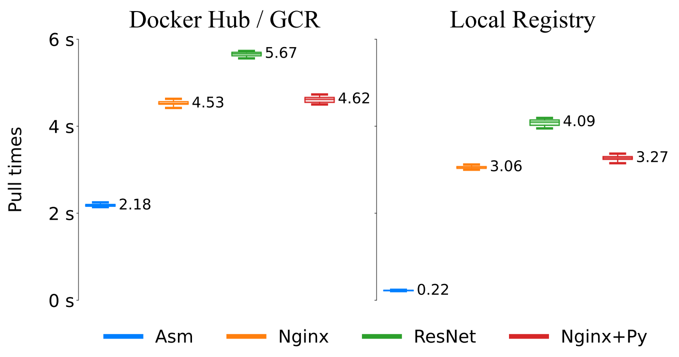

Fig. 13. Total time to *pull (download)* the different service container images onto the *Edge Gateway Server (EGS)* from *Docker Hub* or – in the case of the ResNet container – from *Google Container Registry*. Alternatively, we pull from a private container registry located in the same network.

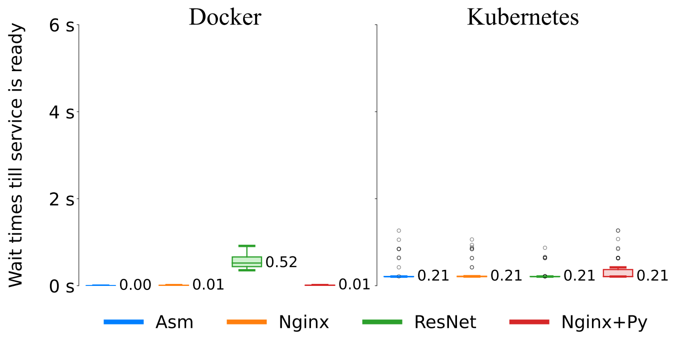

Fig. 14. Wait time (median) until the services are ready after being *scaled up* on two different clusters. Our SDN controller continuously tests whether the respective port is open before setting up the flows. Included in fig. 11.

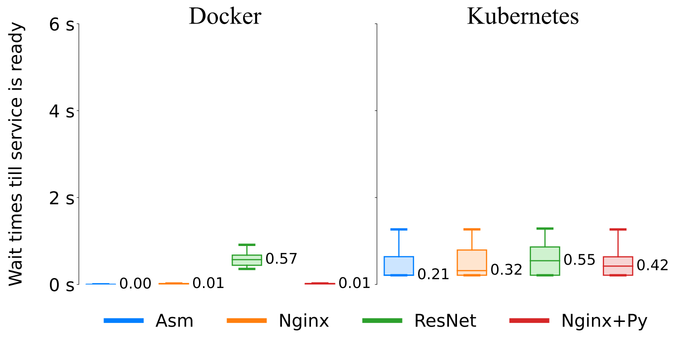

Fig. 15. Wait time (median) until the services are ready after being *created* + *scaled up* on two different clusters. Included in fig. 12.

is where the minuscule *Assembler web server* image shines compared to far larger images like *Nginx*. Note that images are built up of layers; pull times depend on both the image's total size and its number of layers to be downloaded and verified. Furthermore, popular base layers of the image might also be included in other cached images and thus already be on disk.

Once the new service instance is up and running, the benefits of running applications at the edge show. Fig. 16 presents the total time for requests from a client to the edge services once the service instance is up and running on the cluster. As expected, here we see no notable difference between the two clusters. However, while serving a short response message is achieved in about a millisecond, the heavyweight image classification service (*ResNet*) requires significantly longer.

VII. Discussion

Our results in the previous section show that delaying the client's request for on-demand deployment might be feasible for many use cases. Response times of less than one second (with cached Docker images) should be sufficient for all but the most latency-critical applications. And while the significantly higher values of about three seconds might be too much, Kubernetes provides us with automated management and scaling of container instances. However, we can combine the best of both worlds. First, we launch an edge service via Docker to respond faster to the initial request. Then, we deploy the same service to Kubernetes for future requests. This way, we can have both fast initial response (Docker) and automated cluster management (Kubernetes).

In particular, the presented approach is a good option for use cases where a high-bandwidth or locality awareness is more critical than an ultra-low-latency response. Complex applications consisting of multiple microservices are also a good fit. While the initial service is already processing the request, we can still scale up the services later in the workflow. Even many applications that require a low-latency response time might not

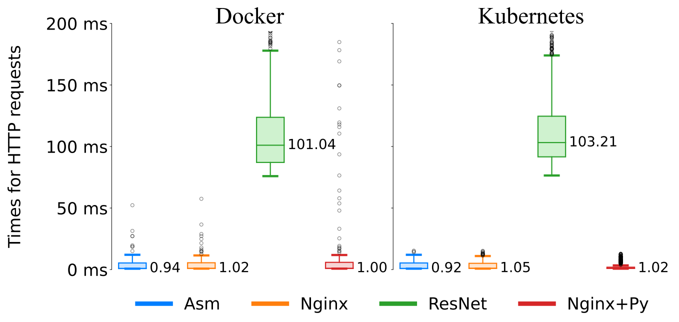

Fig. 16. Total time (median) for client requests to the edge services when the instance is already running on the cluster.

require such a fast response for the initial request. Many of these applications may tolerate a higher response time during the setup phase, requiring a low-latency response only once running. Therefore, on-demand deployment, in combination with transparent access to edge services, provides an excellent fit for many use cases. More so when combined with good prediction for proactive deployment.

VIII. Conclusion and Future Work

In [3], [4], the multiple benefits of transparent access to edge computing services were highlighted, and an efficient approach to implementing such a transparent approach was presented. However, a transparent approach can only work well if the edge services are already running in an edge cluster nearby. In this work, we showed that delaying the client's request for on-demand deployment of containerized services might be feasible for many use cases. Provided that the required container images are locally cached already, response times of less than one second – for the first request – can be achieved when using Docker. This should be sufficient for all but the most latency-critical applications. When deploying to a Kubernetes cluster, it takes significantly longer to start a new service instance – about three seconds – which makes it less suitable for the first request. However, even with applications that require lower response times, an on-demand deployment is a good approach – as long as at least one instance is already running in a nearby edge to process the requests until the new instance is up. In future work, we plan to extend our solution for transparent access by enabling the side-by-side operation of containers and serverless applications and evaluate how well the latter would perform in a transparent access approach.

Acknowledgement

This work received funding from the Austrian Research Promotion Agency (FFG; Grant 888098, Kärntner Fog).

References

[1] D. Kimovski, R. Matha, J. Hammer, N. Mehran, H. Hellwagner, and R. Prodan, "Cloud, Fog, or Edge: Where to Compute?" *IEEE Internet Comput.*, vol. 25, no. 4, pp. 30–36, 2021. [Online]. https://ieeexplore.ieee.org/document/9321525/
[2] ETSI, "MEC in 5G networks," *ETSI White Pap. No. 28*, 2018.
[3] J. Hammer, P. Moll, and H. Hellwagner, "Transparent Access to 5G Edge Computing Services," in *2019 IEEE Int. Parallel Distrib. Process. Symp. Work.* IEEE, 2019, pp. 895–898. [Online]. https://ieeexplore.ieee.org/document/8778343/
[4] J. Hammer and H. Hellwagner, "Efficient Transparent Access to 5G Edge Services," in *2022 IEEE 8th Int. Conf. Netw. Softwarization.* IEEE, 2022, pp. 91–96. [Online]. https://ieeexplore.ieee.org/document/9844066
[5] Y. Miao, F. Lyu, F. Wu, H. Wu, J. Ren, Y. Zhang, and X. S. Shen, "Mobility-Aware Service Migration for Seamless Provision: A Reinforcement Learning Approach," *IEEE Int. Conf. Commun.*, 2022.
[6] "WebAssembly – Binary Instruction Format for a Stack-Based Virtual Machine." [Online]. https://webassembly.org/
[7] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing Serverless to the Edge with WebAssembly Runtimes," *Proc. - 22nd IEEE/ACM Int. Symp. Clust. Cloud Internet Comput. CCGrid 2022*, pp. 140–149, 2022.
[8] "Datadog Container Research Report 2022." [Online]. https://www.datadoghq.com/container-report/ (accessed 2022-12-13).
[9] "Transparent Edge – Transparent Access to Edge Computing Services." [Online]. https://josefhammer.com/r/transparent-edge-repo/
[10] Open Networking Foundation, "OpenFlow Switch Specification v1.5.1," Open Networking Foundation, Tech. Rep., 2015.
[11] "Ryu – Component-Based Software-Defined Networking Framework." [Online]. https://ryu-sdn.org/
[12] T. Taleb, P. Hasselmeyer, and F. G. Mir, "Follow-me cloud: An OpenFlow-based implementation," *Proc. GreenCom-iThings-CPSCom*, pp. 240–245, 2013.
[13] A. Aissioui, A. Ksentini, A. M. Gueroui, and T. Taleb, "On Enabling 5G Automotive Systems Using Follow Me Edge-Cloud Concept," *IEEE Trans. Veh. Technol.*, vol. 67, no. 6, pp. 5302–5316, 2018.
[14] ETSI, "GS MEC 003 - V2.1.1 - Multi-access Edge Computing (MEC); Framework and Reference Architecture," ETSI, Tech. Rep., 2019.
[15] E. Schiller, N. Nikaein, E. Kalogeiton, M. Gasparyan, and T. Braun, "CDS-MEC: NFV/SDN-based Application Management for MEC in 5G Systems," *Comput. Networks*, vol. 135, pp. 96–107, 2018.
[16] A. J. Fahs and G. Pierre, "Proximity-Aware Traffic Routing in Distributed Fog Computing Platforms," *CCGrid*, 2019.
[17] A. Fahs and G. Pierre, "Tail-Latency-Aware Fog Application Replica Placement," *ICSOC 2020 - 18th Int. Conf. Serv. Oriented Comput.*, 2020.
[18] A. J. Fahs, G. Pierre, and E. Elmroth, "Voilà: Tail-Latency-Aware Fog Application Replicas Autoscaler," *Proc. - IEEE Comput. Soc. Annu. Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst. MASCOTS*, 2020.
[19] T. Quang and Y. Peng, "Device-driven On-demand Deployment of Serverless Computing Functions," in *IEEE Int. Conf. Pervasive Comput. Commun. Work.*, 2020.
[20] "AWS IoT Greengrass." [Online]. https://aws.amazon.com/greengrass/
[21] A. Zanni, S. Forsstrom, U. Jennehag, and P. Bellavista, "Elastic Provisioning of Internet of Things Services Using Fog Computing: An Experience Report," *Proc. - 6th IEEE Int. Conf. Mob. Cloud Comput. Serv. Eng. MobileCloud*, pp. 17–22, 2018.
[22] H. Watanabe, R. Yasumori, T. Kondo, K. Kumakura, K. Maesako, L. Zhang, Y. Inagaki, and F. Teraoka, "ContMEC: An Architecture of Multi-access Edge Computing for Offloading Container-Based Mobile Applications," *IEEE Int. Conf. Commun.*, 2022.
[23] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile Cold Starts for Scalable Serverless," *11th USENIX Work. Hot Top. Cloud Comput. HotCloud 2019*, 2019.
[24] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer, "Challenges and Opportunities for Efficient Serverless Computing at the Edge," *Proc. IEEE Symp. Reliab. Distrib. Syst.*, pp. 261–266, 2019.
[25] S. Shillaker and P. Pietzuch, "FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing," *Proc. 2020 USENIX Annu. Tech. Conf. ATC 2020*, pp. 419–433, 2020.
[26] N. Mehran, Z. N. Samani, D. Kimovski, and R. Prodan, "Matching-based Scheduling of Asynchronous Data Processing Workflows on the Computing Continuum," *IEEE Int. Conf. Clust. Comput. ICCC*, 2022.
[27] N. Mehran, D. Kimovski, and R. Prodan, "A Two-Sided Matching Model for Data Stream Processing in the Cloud – Fog Continuum," *Proc. - 21st IEEE/ACM Int. Symp. Clust. Cloud Internet Comput. CCGrid*, 2021.
[28] "asmttpd - Web server written in amd64 assembly (v0.4.5)." [Online]. https://josefhammer.com/r/asmttpd
[29] "Tcpreplay: Sample Captures." [Online]. https://tcpreplay.appneta.com/wiki/captures.html (accessed 2023-02-07).
[30] "Timecurl – Measure HTTP Request/Response Times using Curl." [Online]. https://josefhammer.com/r/timecurl/