Availability and Latency Aware Deployment of Cloud Native edge Slices

Sagar Arora, Adlen Ksentini, Christian Bonnet
Eurecom, Sophia Antipolis, France
firstname.lastname@eurecom.fr

***Abstract*—Edge computing is one of the key technology of the last decade, enabling several emerging services beyond 5G (e.g., autonomous driving, robotic networks, Augmented Reality (AR)) requiring high availability and low latency communications. While in cloud native paradigm, highly embraced by cloud providers, network functions and applications are decomposed into microservices run in a container. Defacto container orchestration engine, namely Kubernetes, deploys multiple containers inside a pod. The mapping between microservices and pod highly affects the availability and latency of deployed microservices and hence the run application. In this paper, we propose novel availability and latency-aware deployment models for an edge service composed of multiple applications designed as multiple microservices. The two considered deployments are analyzed and evaluated using experimentation and an analytical model, considering critical performance criteria for edge-oriented services, like availability and latency requirements.**

***Index Terms*—Cloud-native, Microservices, MEC, CNF, Deployment**

I. Introduction

Edge computing is a critical enabler for ultra Reliable Low Latency (uRLLC) network services [1]. The strict latency and availability demands of the application and network functions can be achieved by placing them at the edge of the network. Besides low latency capability, edge computing reduces the amount of traffic to be transported all the way to the central cloud. Indeed, with edge computing, traffic can be treated locally, keeping data privacy in case of machine learning model training using data collected from sensors and actuators or other elements.

Meanwhile, with the advent of containerization, which led to the emergence of the cloud native principles, applications are no longer deployed as a monolithic block but rather as loosely coupled microservices. In the cloud native world, each microservice is deployed as a container and managed using container orchestration engines and platforms, like Kubernetes. However, edge deployment has still not embraced this trend. Indeed, the key standard of edge computing is ETSI Multi-access Edge Computing (MEC) [2]. The ETSI MEC group has issued several specifications covering: application packaging, orchestration, and traffic redirection; but still considering monolithic blocks when deploying applications at the edge. ETSI MEC considers that applications are deployed in Virtual Machines (VM) or containers but one container or VM per application. All the process of orchestration and management relies on this assumption, which is no longer a reality with the emergence of cloud native orchestration platforms, highly favoring the usage of microservices. To overcome this situation we proposed in [3] a new edge slice orchestration framework that allows describing multiple applications using multiple microservices interacting with each other. In the same paper [3] we devised a novel template, namely Edge Sub Slice Template (ESST), aiming at defining an edge slice containing multiple applications designed using multiple microservices. This template eases the orchestration of containers and microservices on industry defacto cloud native orchestration platform Kubernetes [4].

Basically, to deploy containers Kubernetes uses Pod. The latter is the smallest schedulable entity. It provides an ecosystem for multiple containers to interact. All the Kubernetes-based platforms run containerized microservices in pods. The latency between microservices and their availability depends on whether they are running inside the same pod or different pods and pods are deployed on the same machine or different. The most preferred way to deploy microservices is one pod per microservice i.e., a 1 to 1 mapping between microservice and pod. But, it is also possible to deploy multiple microservices inside one pod, i.e., 1 pod and N microservice inside it. This can be considered as the default solution adopted in [3]. In this context, it is important to understand what is the appropriate solution when considering edge application constraints (i.e., latency, availability, etc.), but also the cost induced to the orchestration and management. The choice of the two possible mappings could be critical if it is not well investigated.

In this paper, we fill this gap by studying the performance of both solutions in terms of availability, latency, and management cost. We combined analytical models and experimentation to quantify the latency and availability metrics. We described such an edge slice using our previously proposed ESST. The paper's contributions are:

- A detailed deployment model for mapping microservices of an edge slice inside one pod and re-modeling the one to one deployment model by classifying microservices as critical and non-critical,
- Markov Chain based availability model for each deployment,

This work was partially supported by the European Union's Horizon 2020 Research and Innovation Program under the 5G!Drones project (Grant No. 857031).

- Latency analysis between microservices for each deployment model.

We used the platform measurements taken on two different testbeds and the availability model to evaluate the availability KPI and latency, respectively. Testbed1 is Eurecom 5G trial facility deployed in Eurecom, Sophia Antipolis, while Testbed2 is OVH[1] Public cloud.

II. Background

Microservices promote flexibility, scalability, and agility over monolithic software design. Cloud native microservice deployment has been recently discussed in ETSI NFV, introducing a new model known as Container Network Function (CNF). However, the ETSI MEC group does not mention microservices-based architecture. Hence, it can only be used to orchestrate a monolithic MEC Application based on VM or container. In contrast, the ESST model we have introduced in [3] and shown in Fig. 1 provides the possibility to design complex microservices-based MEC applications.

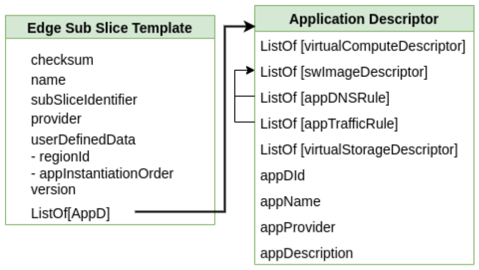

Fig. 1: Edge Sub Slice Template

ESST includes a list of modified versions of MEC Application Descriptors(AppD) [5] to allow orchestrating multiple applications containing multiple microservices on a cloud native orchestration platform. In [3], we introduced the Edge Sub Slice Orchestrator, which receives ESST and parses each AppD, mapping each MEC application to a Kubernetes Pod and microservices constituting the application to containers. An ESST comprising M MEC applications and N microservices will have M pods and N containers as per our orchestration modeling.

To recall, Kubernetes pod is a group of co-located containers. They share the same network namespace, separate computational resources, and can have shared or common storage. The pods are useful for running multiple co-located processes with a degree of isolation, whereas containers are built to run only a single process. The containers inside a pod can communicate with each other using a local network, shared memory IPC (inter-process calls), and shared volumes if the volumes are common.

Disruptions in one container of a pod do not affect the health of other containers inside that pod. Hence, if a container crashes, the pod will be available with other containers, but if a pod crashes, all the containers will crash together. In Kubernetes, to achieve automatic handling of the pod life cycle, it is preferred to deploy them using controller objects. Controllers are responsible for maintaining the desired state of the pod with the observed state. In case of a pod failure, they recreate the pod, and in case of container failure, it is recreated by Kubelet, which is a Kubernetes agent running on each node hosting pods. It is responsible for handling the container life cycle on a node.

In Kubernetes-based platforms, Container Network Interface (CNI) plugin [6] is responsible for creating pod interfaces, IP-address allocation, network security enforcing, and imposing QoS policies. Each CNI plugin uses different network models, for example, underlay and overlay, to create a virtual network on top of a physical network. Every CNI is designed to handle intra-node and inter-node communication differently. The performance of inter pod communication (latency, throughput, bandwidth, etc.) depends on the network model and type of tunneling protocol (e.g., VXLAN, IP-in-IP or Geneve, etc.) used by CNI. In [7] the authors have assessed the performance of various CNI plugins in terms of functionality, performance, and scalability. They explain how different plugins interact with the node's network stack and how increasing the number of pods affects the inter-pod communication.

III. Deployment Models for edge Slice

In the formation of an ESST, the number of MEC Applications and microservices depends on the edge slice template provider or MEC Application provider. The enablers of edge slice functionality are microservices; together, they deliver MEC Application's desired features. Hence, we can consider a ESST composed of microservices mapped to containers, and these containers can be placed inside the same pod or different pods. These pods can run on the same machine or different machines.

Based on this, we consider two different deployment models for an ESST with N microservices. Fig. 2 shows a 1 to 1 mapping between MEC Application and microservices, one container per pod. The edge slice will have N MEC applications or pods and N microservices or containers. Fig. 3 shows 1 to N mapping between MEC Application and microservices, i.e., N containers in one pod. The slice will have 1 MEC application or pod and N microservices or containers.

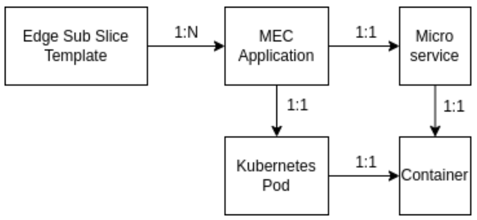

Fig. 2: 1 to 1 Mapping of MEC Applications and Microservices

In the following sub-sections, we will analyze and discuss each deployment model, its benefits, and drawbacks considering critical performance criteria: (i) Orchestration and management, (ii) Availability, and (iii) Low-Latency.

[1]https://www.ovhcloud.com/en/public-cloud/kubernetes/

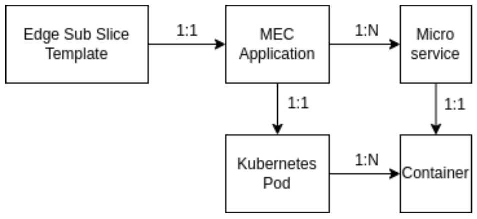

Fig. 3: 1 to *N* Mapping of MEC Application and Micro-services

A. Orchestration and Management

The process of orchestration and management is one of the key functions in the context of containerization. It concerns the life cycle management (LCM) of a network slice and hence all the applications that constitute the network slice. The most time-consuming task in the life cycle management of a slice is slice creation. It involves fetching container images of the microservices, scheduling, computational and storage resource allocation, IP-address allocation, instantiation, etc.

The number of container images remains the same for both deployment models. Containers mostly consume the computational and storage resources, and in both deployment models, the number of containers remains the same, so the computational and storage resource consumption will be the same. However, if there is a significant pod overhead[2] then the 1 to N deployment model will use fewer resources than 1 to 1. Each pod is allocated an IP-address; hence 1 to 1 deployment model will have N IP-addresses, whereas 1 to N will have only one IP-address.

Pod scheduling and instantiation are the two most critical stages. They result in variable creation time for both the deployment models. In pod scheduling, the scheduler has to look for host machines that have enough computational resources for these pods. In the 1 to N model, all the containers stay in the same pod. Hence, if there is no such machine with requested computational resources, it will result in scheduling failure. In contrast, the 1 to 1 model can schedule pods on different nodes; hence the computational resources can be utilized efficiently. Once the pods are scheduled, their Cgroups, network interfaces, etc., are created. This step's time depends on whether the pods are instantiated in parallel or serial (if pods require a particular order of instantiation). The containers inside a pod only start parallel. Thus, if the pods are started serially, the 1 to N deployment takes less time, and if pods are started in parallel, the 1 to 1 deployment takes less time.

B. Availability

Availability is a critical component for any type of telecom and vertical service, especially when the service belongs to the uRLLC category. It measures the length of time a system or network is functioning. In carrier-grade systems, the availability should ensure the 5 nine uptime, i.e., 99.999% available. To ensure availability in cloud-native systems, it is important to reduce the downtime and guarantee that all microservices, and hence pods, are active. The different reasons that lead to reduced availability are

- The container running the microservice died due to an internal error (software bug, high resource consumption, etc.) in the microservice.
- The pod died, and hence all the containers inside the pod also died.
- The node hosting the pod died due to hardware or software errors.

The reasons for the above failure can be voluntary or involuntary. In case of a node failure, pods with their containers are immediately scheduled on another node. Therefore, a slice will be disrupted for the time the pods are getting instantiated.

If we consider the first deployment model, i.e., 1 to 1, we expect better availability. Indeed, if the container or the pod died, only one microservice died, and the time to reboot 1 pod with 1 container is very short. However, the pod needs to be re-scheduled on another node if the node dies. If another node is available, then the time to reboot the pod is short. Otherwise, the downtime can be higher. In the second deployment model (1 to N), when a pod died, all the microservices running inside the pod died with it. This means that the time needed to reboot the pod with all its containers depends on the number of containers to reboot. In the 1 to 1 deployment model, the pods can be scheduled across different nodes; even if some pods are not available, the edge slice will be partially available. Whereas, in the other model, if there is any disruption, the edge slice will not be available completely; all the containers are running in the same pod.

C. Latency

As stated earlier, latency is the critical KPI that motivates the deployment of cloud native microservices at the edge. In cloud native and virtualized systems, we distinguish the communication latency among the microservices and the service latency (i.e., the collective functionality of all the microservices). The service latency is composed of inter microservice latency and microservices processing time. Service latency may increase due to the downtime if a container, a pod, or a node dies. Regarding the communication latency, we can assume that the 1 to N model will achieve the best performance as all the microservices are inside the pod and use the pod's loopback interface or Inter Process Calls (IPC) via Unix sockets that ensure merely instantaneous communications among microservices.

In contrast, the 1 to 1 deployment model may have a communication overhead, particularly if pods are deployed on different nodes, which requires packets to traverse through tunnels, which negatively impacts communication latency. One solution that can improve the performance of the 1 to 1 model is to use a placement algorithm that ensures that all pods are grouped in the same node, which avoids using tunnels and hence reduce communication latency. If the node hosting all these pods dies, the whole slice will not be functional, impacting the availability strongly. Another solution would be to

[2]https://kubernetes.io/docs/concepts/scheduling-eviction/pod-overhead/

use special CNIs that use Datapath Acceleration Development Kit (DPDK) [8] in combination with Single-root input/output virtualization (SR-IOV) to improve latency and throughput metrics. Most of the edge providers may choose this option to enable low latency demanding services.

Concerning the service latency, and as discussed in the availability section, 1 to 1 deployment model should achieve better results in comparison to 1 to N as it minimizes the downtime.

IV. Modeling Availability

To evaluate the availability of both the deployment models, we proposed to model them using Markov Chains. Let us assume that

- An ESST is composed of N microservices
- Each microservice in a slice is categorized into critical and non-critical. The non-critical ones provide extra functionality/features to the slice. The critical ones are responsible for the primary/principal functionality of the slice.
- If a non-critical microservice dies, the edge slice will be partially disrupted. Indeed, it can still work with limited functionality. But, if a critical microservice dies, the edge slice will be completely disrupted.
- The microservices have no software-related bugs, and the computational resources needed by the microservices are properly allocated to their container. This avoids the failure of the microservice container due to internal errors. We will only consider the failure of the pod enclosing microservice container.
- We assume that the failure rate of pods and recreation rate of pods follows an exponential distribution with rates of f and r, respectively.

Let us denote M and K by the number of non-critical and critical microservices, respectively. Here, $N = M + K$.

Fig. 4 represents the slice deployed using the 1 to N deployment model, where N microservices or containers are inside one pod. Here, due to the compact nature of the deployment, if the pod fails, all the containers will fail. Besides, we don't differentiate between critical and non-critical microservices. We assume that the recreation period of a pod is composed of the time needed to recreate all the containers, which can be modeled using an exponential distribution with a rate r_{max}. Similarly, the failure rate can be modeled using an exponential distribution f_{max}. Therefore, we model the system with a two-state Markov chain $X = X(t), t \geqslant 0$ on the two states 0 and 1; where 0 indicates that the system is in failure, while 1 means that the system is fully available. A transition between state 0 to 1 with the rate r_{max} indicates that the pod is recreated, and a transition between state 1 to 0 with the rate f_{max} indicates that the system has failed. Fig. 4 illustrates the transition graph.

In this scenario, the time a slice with N microservices will be available corresponds to the probability to be in state $S = 1$ denoted by,

$$A = r_{max}/(r_{max} + f_{max}) \quad (1)$$

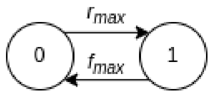

Fig. 4: Markov chain for 1 to N deployment model

Regarding the other deployment model 1 to 1, we define a Markov chain $X = X(t), t \geqslant 0$ on the state space S defined by $S = \{(m,k)|m = 0, ..., M \text{ and } k = 0, \ldots, K\}$, for every $K \geqslant 1$. In this model, $X(t) = (m,k)$ indicates that, at time t, there are m active non-critical microservices and k active critical microservices. While $s = (0,0)$ indicates that all the pods are down, $s = (M,K)$ indicates that all pods are healthy and work properly. Fig. 5 illustrates the transitions graph of the envisioned system.

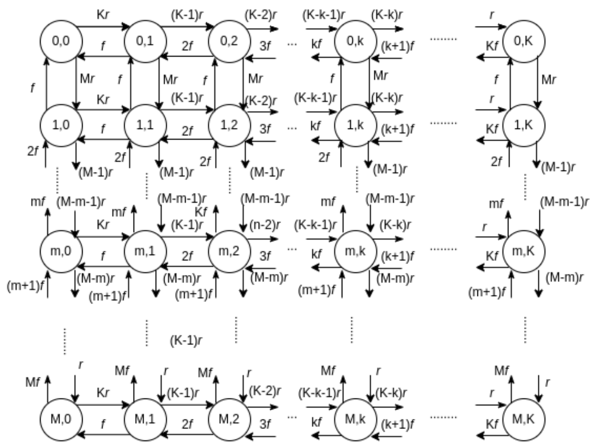

Fig. 5: Markov Chain for 1 to 1 deployment model

- If a non-critical pod gets recreated while already m $(0 \leqslant m \leqslant M-1)$ are active and k critical containers are active then there is a transition from state (m,k) to state $(m+1,k)$ with rate $(M-m)r$.
- If a critical pod gets recreated while already k $(0 \leqslant k \leqslant K-1)$ are active and m non-critical containers are active then there is a transition from state (m,k) to state $(m,k+1)$ with rate $(K-k)r$.
- If a non-critical pod fails when m $(0 \leqslant m \leqslant M-1)$ are active and k critical containers are active then there is a transition from state (m,k) to $(m-1,k)$ with rate $(m)f$.
- If a critical pod fails when k $(0 \leqslant k \leqslant K-1)$ are active and m non-critical containers are active then there is a transition from state (m,k) to $(m,k-1)$ with rate $(k)f$.

Let Q be the infinitesimal generator matrix for the chain. Each entry q_{mk} such as $s_m, s_k \in S$ and $m \neq k$ of the matrix corresponds to the instantaneous transition rate from state m to state k. Diagonal entries are chosen to ensure null rows of Q, i.e.:

$$q_{mm} = -\sum q_{mk}, s_k \in S, m \neq k \quad (2)$$

The objective is to analyze the system in long run; i.e., inter-events time are neglected over the running time of the system.

It matches the steady state behavior of the analyzed system. $\forall s_m \in S$, we note $\pi_m = \pi_{mk} = \lim_{t\to\infty} P\{s(m,k)\}, m \in (0,M), k \in (0,K), M+K = N$, the stationary probability distribution of the chain. The Markov chain of 1 to 1 is homogeneous, finite and irreducible process. In the steady state of the system, we assume that the total probability flux out of a state is equal to the total probability flux into the state. For a state $s_m \in S$:

$$\pi_m * \sum_{s_k \in S, m \neq k} q_{mk} = \sum_{s_k \in S, m\neq k} \pi_k * q_{mk} \tag{3}$$

Let π be the vector containing all model states. By combining 2 and 3, we can write

$$\pi * Q = 0 \tag{4}$$

The normalization condition of the chain is

$$\sum_{s_k \in S} \pi_m = 1 \tag{5}$$

Solving global balance and normalization condition equations 4 and 5 leads to determine vector π. Getting the steady-state probabilities will allow us to determine the probability of having m non-critical pods and k critical pods active. This, in turn, will help to derive the availability of the 1 to 1 deployment model.

The slice applications are fully available when it is in the state $s = (M,K)$ i.e., all pods are running, it is denoted by A_{full}

$$A_{full} = P(s(M,K)) \tag{6}$$

The slice applications are available with limited capabilities/functionalities when it in the state $s = (m,K), \forall m \in (0,M)$

$$A_{limited} = \sum_{m=0}^{M} P(s(m,K)) \tag{7}$$

In special scenarios where it is not possible to distinguish between critical and non-critical, i.e., all microservices are critical, we can use the Markov chain corresponding to the 1 to N model. The slice will be available when all the pods are running.

V. Performance Evaluation

Latency and availability KPIs are highly dependent on the design of the infrastructure, computational resources of the node hosting pods, and connectivity between the nodes. To evaluate the performance of the deployment model irrespective of the infrastructure, we choose two Kubernetes managed cloud environments to analyze and evaluate our proposed deployment models. The testbed1 is a 5G trial facility deployed in Eurecom. It has a production-grade Openshift Cluster with 5 bare metal worker nodes 212 CPU cores, and 372 GiB of RAM connected via Openshift SDN-based CNI. The testbed2 is a managed Kubernetes service rented from public cloud OVH. It has 5 virtual machine-based worker nodes with 2 vCPU and 7 GiB of RAM connected via Canal CNI.

Fig. 6 shows the Round Trip Time (RTT) to understand the latency between microservices when they are deployed with two different models. We used Google Remote Procedure Call (GRPC), one of the most used protocols in microservices architecture, Inter Process Communication (IPC) over Unix socket, and GRPC over Unix socket. The unix socket-based communication can only happen in 1 to N type of deployment model as the microservices share the same network namespace. The * in 1 to 1 deployment model depicts communication between microservices deployed on the same machine and without * on different machines. We also added the results of ICMP ping for readers who specifically use ping as a latency metric. Below conclusion can be drawn from Fig. 6

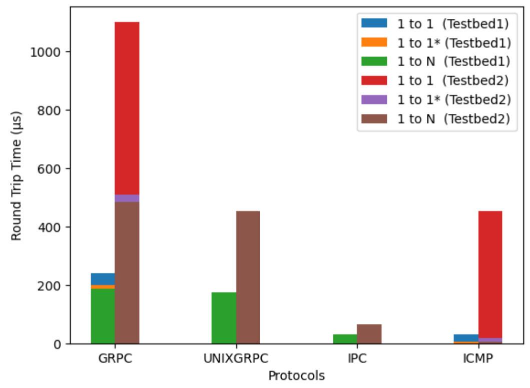

Fig. 6: Inter Microservice RTT (μS) for different models and testbeds

- Different RTT values in two testbeds is due to different CNIs, computational and network resources associated with the cluster.
- In both the testbeds 1 to N deployment model has the lowest RTT as it uses loopback interface or UNIX sockets. Accordingly, it is better to use IPC based for latency-sensitive applications.

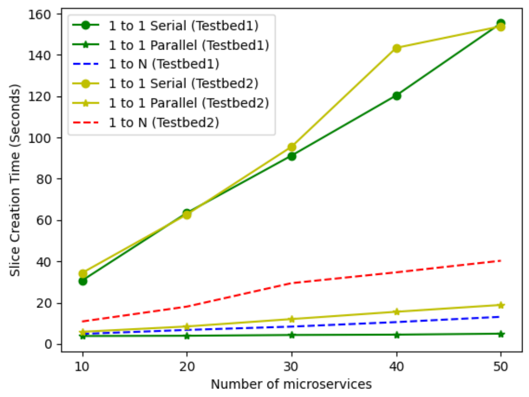

Fig. 7: Slice Creation Time (S) for different models and Testbeds

We created slices with 10, 20 up to 50 microservices on both testbeds to evaluate the slice creation time. We used the same container image for all the microservices and set the resource requirement at 10 milliCPU and 100Mb Ram. The slice with 1 to 1 deployment model can be created with or without dependency between the microservices, i.e., serial or parallel pod creation. Fig. 7 shows the slice creation time. We conclude that,

- 1 to 1 model takes less time when the microservices do not have any dependency among them, i.e., pods are created parallelly.
- 1 to N deployment model in testbed1 takes less time than 1 to 1 in testbed2. This behavior is due to the different computational capacities of the two clusters.

The pods were deployed using Kubernetes deployment controller, which watches for the pod availability. If the pod is not running, it immediately creates a new pod. We forcibly deleted the pods to simulate the pod recreation in the 1 to 1 and 1 to N deployment model. Fig. 8 shows the average pod recreation time with the increasing number of microservices. We assume that microservices are again available once the pods are recreated.

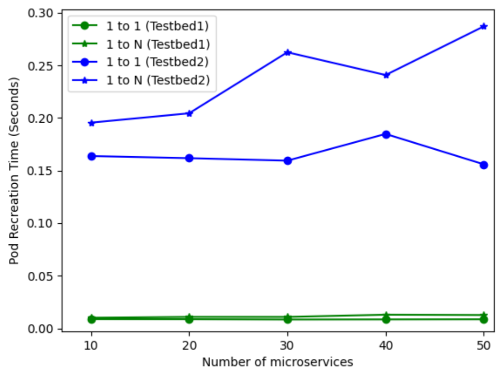

Fig. 8: Pod Recreation Time for different models and Testbeds

We use these values of pod recreation time to estimate the availability for a slice with 10 and 50 microservices deployed with 1 to 1 and 1 to N deployment model. We assume that the pod failure rate is 100 times in a year, and it can happen at any point in time. In table I, we calculated full availability of 1 to 1 and 1 to N deployment models using the formula derived via solving the Markov Chains, equation 6 and 1 respectively. To correlate availability with a number of critical and non-critical microservices, we calculated availability for a slice with 50 microservices twice as depicted in scenarios II and III.

TABLE I: Availability of a Slice with 10 and 50 Microservices

Scenario	Testbed	Critical	Non-Critical	1 to 1 (Full)	1 to N
I	Testbed1	5	5	99.9998%	99.9999%
I	Testbed2	5	5	99.997%	99.9992%
II	Testbed1	25	25	99.999%	99.9999 %
II	Testbed2	25	25	99.9856%	99.9989%
III	Testbed1	40	10	99.9986%	99.9999%
III	Testbed2	40	10	99.9765%	99.9989%

We can draw the following conclusions from Fig. 8 and table I,

- Computational resources of the testbed affect the pod recreation time, which indeed affects the slice availability
- From scenarios II and III, we can see the effect of critical microservices on the slice availability. If the majority of the slice microservices are critical, then there are more chances of a critical microservice getting failed. Hence, the availability will reduce.

The 1 to N deployment model has higher availability than the 1 to 1 model, but this is highly subjective to the availability of a node with required computational requirements. 1 to 1 deployment has an advantage over 1 to N, which is the limited availability. If a non-critical microservice fails, the slice will still be available with limited functionality. In contrast, in 1 to N deployment, the slice will be completely non functional until a suitable node is found.

VI. Conclusion

In this paper, we presented a novel methodology to model a cloud native network slice defined with microservices. The two deployment models and our approach of classifying microservices into critical and non-critical are suitable for different types of applications. Both deployment models are useful in serving different purposes. The 1 to N deployment model promises low latency communication and high availability with the condition that the computational resources required by the microservices are available all the time in one of the cluster nodes. It might not be the case all the time. In the 1 to 1 deployment model, availability is subjected to the application's design in terms of the number of critical and non-critical microservices. This model efficiently utilizes clusters computational resources by allowing microservices to spread across different cluster nodes. It is suitable to use this model for applications prioritizing availability over low latency and can be categorized into critical and non-critical microservices.

References

[1] A. Ksentini et al, "Providing low latency guarantees for slicing-ready 5G systems via two-level MAC scheduling," in IEEE Network, vol. 32, no. 6, 2018.

[2] B. Brik et al,"Service-oriented MEC applications placement in a federated edge cloud architecture", in Proc. of IEEE international conference on communications (ICC), Virtual, 2020.

[3] S. Arora, et al, "Lightweight edge Slice Orchestration Framework," in IEEE ICC 2022, May 2022.

[4] B. Burns, et al, 2016. "Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade". Queue 14, 1 (January-February 2016), 70–93.

[5] Multi-access Edge Computing (MEC); MEC Management; Part 2: Application lifecycle, rules and requirements management,ETSI GS MEC 010-2 V2.2.1 Feb. 2022.

[6] CNI—Container Network Interface. Accessed: Apr. 26, 2022.[Online]. Available: https://github.com/containernetworking/cni

[7] S. Qi, et al, "Assessing Container Network Interface Plugins: Functionality, Performance, and Scalability," in IEEE Transactions on Network and Service Management, vol. 18, no. 1, pp. 656-671, Mar. 2021.

[8] DPDK- Data Plane Development Kit. Accessed: Apr. 26, 2022.[Online]. Available: https://www.dpdk.org