Latest updates: https://dl.acm.org/doi/10.1145/3634750

PDF Download
3634750.pdf
17 March 2026
Total Citations: 20
Total Downloads: 2705

Published: 14 February 2024
Online AM: 04 December 2023
Accepted: 01 November 2023
Revised: 31 October 2023
Received: 05 November 2022

Citation in BibTeX format

RESEARCH-ARTICLE

NEPTUNE: A Comprehensive Framework for Managing Serverless Functions at the Edge

LUCIANO BARESI, Politecnico di Milano, Milan, MI, Italy

software engineering, cloud computing, edge computing, self-adaptive systems, mobile apps.

DAVIDE YI XIAN HU, Politecnico di Milano, Milan, MI, Italy

GIOVANNI QUATTROCCHI, Politecnico di Milano, Milan, MI, Italy

LUCA TERRACCIANO, Politecnico di Milano, Milan, MI, Italy

Open Access Support provided by:

Politecnico di Milano

ACM Transactions on Autonomous and Adaptive Systems, Volume 19, Issue 1 (March 2024)
https://doi.org/10.1145/3634750
EISSN: 1556-4703

NEPTUNE: A Comprehensive Framework for Managing Serverless Functions at the Edge

LUCIANO BARESI, DAVIDE YI XIAN HU, GIOVANNI QUATTROCCHI, and LUCA TERRACCIANO, Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Italy

Applications that are constrained by low-latency requirements can hardly be executed on cloud infrastructures, given the high network delay required to reach remote servers. Multi-access Edge Computing (MEC) is the reference architecture for executing applications on nodes that are located close to users (i.e., at the *edge* of the network). This way, the network overhead is reduced but new challenges emerge. The resources available on edge nodes are limited, workloads fluctuate since users can rapidly change location, and complex tasks are becoming widespread (e.g., machine learning inference). To address these issues, this article presents *NEPTUNE*, a serverless-based framework that automates the management of large-scale MEC infrastructures. In particular, *NEPTUNE* provides (i) the placement of serverless functions on MEC nodes according to users' location, (ii) the resolution of resource contention scenarios by avoiding that single nodes be saturated, and (iii) the dynamic allocation of CPUs and GPUs to meet foreseen execution times. To assess *NEPTUNE*, we built a prototype based on K3S, an edge-dedicated version of Kubernetes, and executed a comprehensive set of experiments. Results show that *NEPTUNE* obtains a significant reduction in terms of response time, network overhead, and resource consumption compared with five state-of-the-art solutions.

CCS Concepts: • **Theory of computation** → *Scheduling algorithms*; • **Computing methodologies** → *Distributed computing methodologies*; • **Computer systems organization** → *Distributed architectures;*

Additional Key Words and Phrases: Serverless, edge computing, GPU, placement, vertical scaling, kubernetes, k3s, dynamic resource allocation, control theory

ACM Reference format:
Luciano Baresi, Davide Yi Xian Hu, Giovanni Quattrocchi, and Luca Terracciano. 2024. *NEPTUNE*: A Comprehensive Framework for Managing Serverless Functions at the Edge. *ACM Trans. Autonom. Adapt. Syst.* 19, 1, Article 7 (February 2024), 32 pages.
https://doi.org/10.1145/3634750

1 INTRODUCTION

Multi-access Edge Computing (**MEC**) [19, 38] has been presented as a new highly distributed architecture to run applications at the edge of the network and close to users' locations. The

This work has been partially supported by the SISMA national research project (MIUR, PRIN 2017, Contract 201752$ENYB$) and by the EMELIOT national research project (MIUR, PRIN 2020, Contract 2020$W3A5FY$).

Author's address: L. Baresi, D. Y. X. Hu, G. Quattrocchi, and L. Terracciano, Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, via Golgi, 42 20133 Milano (Italy), Milan, Italy; e-mails: luciano.baresi@polimi.it, davideyi.hu@polimi.it, giovanni.quattrocchi@polimi.it, luca.terracciano@polimi.it.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1556-4665/2024/02-ART7 $15.00
https://doi.org/10.1145/3634750

network delay needed to reach servers is zeroed and the overall execution time is significantly reduced. Unlike cloud frameworks, which provide the abstraction of "infinite" resources, MEC infrastructures are characterized by a set of geo-distributed, resource-constrained nodes (e.g., 5G base stations).

These infrastructures allow one to design applications with strict requirements on response time such as autonomous driving [35], VR/AR [10], and mobile gaming [63] systems. Li et al. [29] measured that the average network delay to reach EC2 Amazon servers from 260 random user locations is approximately 74 ms. Such overhead could make strict latency requirements of a few hundred milliseconds [31] nearly impossible if the application is executed on the cloud [25]. In tasks like obstacle detection for autonomous driving, the overall latency should be minimized and the network overhead of cloud providers may lead to slow responses that could harm the security of users. Recent mobile devices may allow for executing these computations locally to eliminate network latency and preserve user privacy, but this is not always possible: resource consumption is a critical factor (e.g., draining battery) and the complexity of some tasks (e.g., machine learning) may require more powerful hardware.

The traffic generated by edge clients is highly dynamic since users (quickly) move across different locations (e.g., autonomous vehicles) and the intensity of workloads can rapidly increase (e.g., several users that reach the same destination for an event). Cloud infrastructure management relies on autoscaling systems and resources, using techniques like heuristics, machine learning, and time-series analysis [13, 48, 65]. However, these methods rest on the assumption of virtually infinite resources and fast networking, which is not viable in edge computing. Therefore, context-specific alternatives are needed, as demonstrated by Ascigil et al. [1] and Wang et al. [58] for managing resource-limited nodes, and by Poularakis et al. [41] for efficient request routing and load balancing at the edge. In general, solutions that work on computation placement or request routing aim to maximize the throughput of edge nodes, but solutions that address placement, routing, and strict response times at the same time are still in their infancy. What is more, the complexity of computational tasks, like AI-based computations at the edge, is increasing, and more powerful hardware, such as GPUs, is becoming key to speed up these computations [9]. The combined management of CPUs and GPUs is seldom taken into account explicitly [6, 23, 51], while an integrated solution would be desirable.

To tackle all these issues, this article presents *NEPTUNE*, a comprehensive solution for the runtime management of applications that run on a large-scale edge infrastructure. Our approach handles placement, routing, and CPU/GPU allocation in an integrated way for the efficient execution of edge applications. *NEPTUNE* exploits the *serverless* paradigm [22] to allow users to deploy and execute lightweight *functions* without the burden of managing the underlying infrastructure, and **Mixed Integer Programming** (**MIP**), to place functions on nodes close to users' locations, minimize delays and optimize the usage of available resources. *NEPTUNE* favors the allocation of the functions that can be accelerated with dedicated hardware onto nodes that are equipped with GPUs, if available. Finally, it employs control theory to provision CPU cores dynamically and manage rapidly changing workloads.

We only require that users define the functions of interest and foreseen response time. *NEPTUNE* deploys the functions by means of containers, monitors their performance as well as the locality and intensity of workloads, and reacts automatically. Unlike other solutions (see Section 6) that only target one or few aspects of the management of such systems, *NEPTUNE* tackles MEC applications in a coherent way and controls their complete lifecycle: from deployment to placement and runtime management.

In addition to the theoretical model, this article also presents a prototype framework built on top of K3S,[1] a distribution of Kubernetes[2] (the de-facto standard tool for container orchestration) dedicated to edge computing. Our prototype assumes that functions be OpenFaaS-compliant.[3]

To mimic a MEC infrastructure, we created a geo-distributed cluster of virtual machines on **Amazon Web Services** (**AWS**) and evaluated *NEPTUNE* through a set of experiments. We executed a comprehensive set of tests using a popular benchmark for serverless applications and a machine learning application that can be accelerated by GPUs. We also compared our solution against five state-of-the-art approaches. The comparison revealed that *NEPTUNE* obtains 42.5% fewer constraint violations, reduces network delay of 44.5%, and uses fewer resources (50.8% improvement).

This article extends [4] with three main contributions: (i) an extension of the optimization problem that takes into account both delays and cost of resources at the edge (ii) an in-depth description of our prototype, and (iii) three sets of new experiments that compare *NEPTUNE* against both industrial and academic approaches and that evaluate, respectively, the autoscaling features of *NEPTUNE*, how resource contention affects its performance, and three different configurations of the extended optimization problem.

The rest of the article is organized as follows. Section 2 defines the problem addressed by *NEPTUNE* and introduces the solution. Section 3 illustrates how we formalized the placement, routing, and GPU/CPU allocation problems. Section 4 describes our prototype that extends well-known open-source software. Section 5 presents the experiments we carried out. Section 6 surveys the related work, and Section 7 concludes the article.

2 *NEPTUNE*

NEPTUNE enables the execution of multiple, latency-constrained applications on a MEC infrastructure. This means: (a) component placement on MEC nodes, (b) request routing, and (c) resource management/provisioning. At the same time, *NEPTUNE* considers nodes with limited resources, available network bandwidth, fluctuating workload, and tight response times.

A MEC infrastructure allows a set N of nodes, distributed in different areas at the edge of the network, to run a set A of applications. Applications hosted on MEC nodes can be accessed by clients (such as autonomous vehicles, smartphones, or VR/AR equipments). Each node $n \in N$ is equipped with CPU cores, memory, and possibly GPUs. Due to user mobility, the requests targeted to n might ramp up quickly, and limited resources might prevent n to process every request: certain requests must be outsourced to adjacent nodes.

Let r be an incoming request for application $app \in A$, the response time RT to r is calculated as the time needed to (i) send r from the client to the nearest node i, (ii) process the request, and (iii) transfer the response back to the client. Formally, RT is defined as $RT = E + Q + D$, where E (*execution time*) denotes the amount of time required to run a, Q (*queue time*) is the amount of time that r waits to be handled, and D defines the round-trip network delay (or network latency).

Figure 1 shows a MEC topology of four nodes and three running applications: one dedicated to self-driving vehicles (denoted by symbol ○), one to healthcare (denoted by symbol □), and one to virtual reality (denoted by symbol ◇). The colored symbols below each node are the applications running on it, while the rounded rectangles contain the clients connected to that node. The round trip time to outsource an execution from node i to node j is denoted by $\delta_{i,j}$. In the figure, some clients close to Node N1 are requesting application ◇ but an instance of ◇ is not available on the

[1] https://k3s.io
[2] https://kubernetes.io
[3] https://www.openfaas.com/

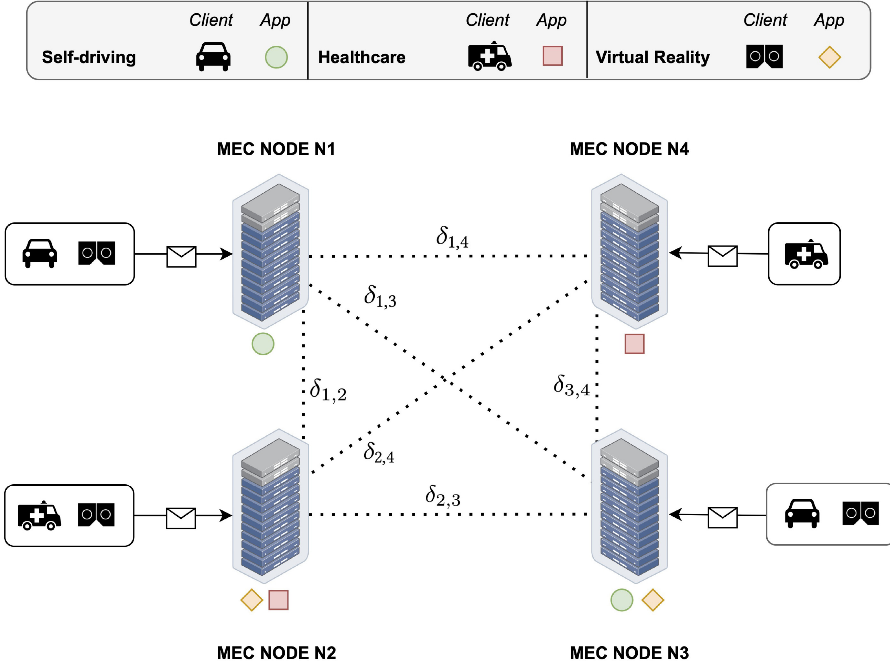


Fig. 1. Example MEC topology.

node. Thus, Node N1 needs either to launch an instance of ◇ or forward the request to Node N2 or N3 that already host application ◇.

NEPTUNE takes into account the geographical distribution of both nodes and incoming traffic and manages requests as soon as they reach the MEC infrastructure. The time a client needs to connect to the nearest node is optimized by existing protocols [11]. *NEPTUNE* tracks where and how many requests are generated by users and exploits an abstraction of the MEC network—generated by measuring inter-node latency— to serve them. Clients can also set a threshold (service level agreement) on the *requested* response time (RT^R_{app}) for each application *app*.

2.1 Solution Overview

NEPTUNE assumes that an application *app* be developed and deployed as a set F_{app} of functions — as envisioned by the serverless paradigm—wrapped in lightweight virtualization technologies like containers. The developer writes the functions, and *NEPTUNE* automatically manages the underlying MEC infrastructure. This way, *NEPTUNE* manages relatively small components that allow for more flexibility and agility while scaling the system compared to traditional (e.g., monolith) and more recent (e.g., microservices) architectures.

For each function f, users must specify a requested response time RT^R_f (with $RT^R_f \leq RT^R_{app}$) and the memory required to properly run it on CPUs (m^{CPU}_f) and on GPUs (m^{GPU}_f), if possible. *NEPTUNE* then automatically deploys f, by creating one or more function *instances*, and manages its operation to guarantee set response time.

To manage the function instances deployed in containers (one instance per container), *NEPTUNE* employs a three-level control hierarchy, namely *Topology*, *Community*, and *Node* levels. A shared monitoring layer lets the different components communicate with each other. This enables the collection of performance metrics, such as response times and network latency, which are needed

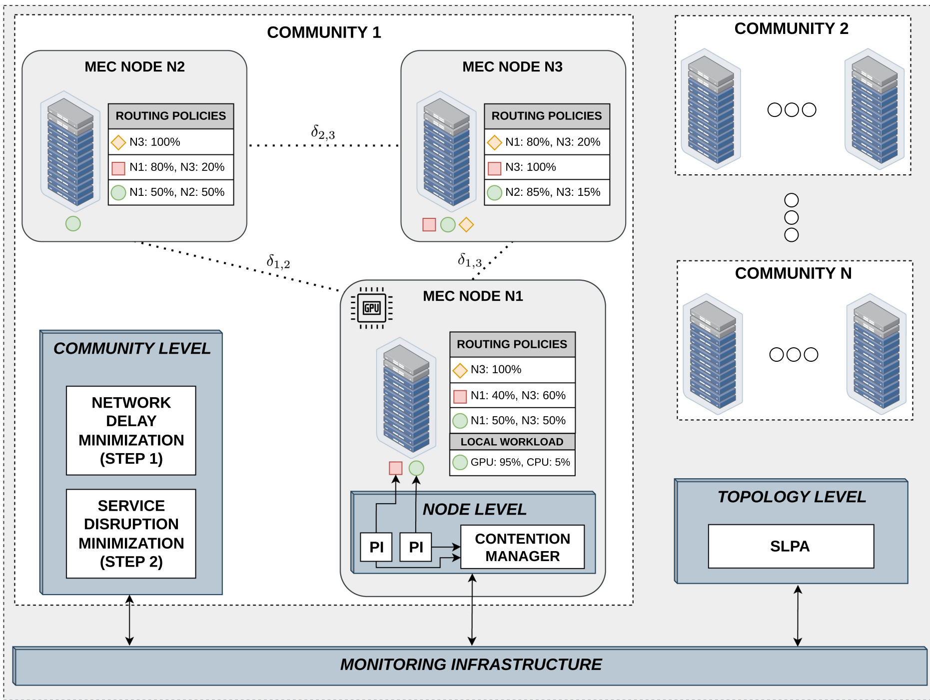


Fig. 2. High-level view of *NEPTUNE*.

by the three control levels to operate correctly and precisely. The Node level and Community level do not require any prior knowledge of the hardware configurations of the different nodes, and they can manage heterogeneous nodes equipped with different types of CPUs/GPUs and with different virtual machines. Figure 2 shows a MEC topology managed by the three-level hierarchical control employed by *NEPTUNE*. The main *NEPTUNE* components are shown as dark grey boxes.

Given that function placement is a well-known NP-hard problem [41], the Topology level reduces the complexity for the other levels by partitioning the MEC topology into *communities* of closely-located nodes. Requests that reach a node in a community are handled within the community itself. If it is not possible, the communities are probably unbalanced and the Topology level must re-partition the topology accordingly. This way, communities are independent of one another.

The Topology level exploits a single controller that is based on an existing clustering algorithm named ***Speaker-listener Label Propagation Algorithm*** (**SLPA**), which was originally presented by Xie et al. [61]. The complexity of SLPA is $O(t * N)$ where N is the number of MEC nodes and t is the maximum allowed number of iterations. Being the computational complexity linear with the number of nodes, this solution is practically usable even for large MEC infrastructures [5]. SLPA partitions the topology in a set of communities with a maximum size of *MCS* nodes and a maximum inter-node delay Δ (i.e., $\delta_{i,j} \leq \Delta$ for all nodes i and j in the community). SLPA allows a single node to be shared among multiple communities. To prevent resource contention and to avoid dependencies among communities, *NEPTUNE* re-allocates shared nodes—if needed—to generate non-overlapping communities. Figure 2 shows a MEC network with N communities.

At Community level, a controller oversees the management of function instances. *NEPTUNE* exploits a dedicated controller for each community, and there is no direct communication among

them. The controller is in charge of placing function instances on the community's nodes. Given the Node level controllers are able to vertically scale function instances, no more than one instance per function type can be deployed on a single node. Given the combinatorial nature of the placement problem, the Community level controller exploits a formulation based on MIP. This formulation supports two types of functions: the ones that can be run on both GPUs and CPUs (e.g., the inference phase of machine learning applications) and the ones that can only run on CPUs. The optimization process first considers the availability of GPUs for functions that support them and then allocates the remaining functions to traditional CPUs. The objective of the controller is to minimize, at the same time: (i) network delay by placing function instances as close as possible to where the workload is generated, and (ii) number of MEC nodes required to serve the workload. However, there exists a tradeoff between these two factors since fewer nodes may limit the ability to place function instances close to the workload. To address this inherent tradeoff, *NEPTUNE* provides users with the flexibility to prioritize one of the factors according to their specific requirements.

The Community level also computes a set of routing policies to permit each node to forward a well-defined and deterministic percentage of the workload to other nodes. If a node is equipped with GPUs and runs one or more computation-intensive applications, the set of routing rules also defines the percentage of local requests that are handled by GPUs and those executed on CPUs. *NEPTUNE* always tries to fully exploit the GPUs and uses CPUs only when they are saturated.

By computing placement actions, the Community level can add or remove function instances on the MEC nodes (horizontal scaling). For example, if the resources of node i are not sufficient to handle the entire workload $\lambda_{f,i}$ of function f, the Community level creates a new instance (of f) on a node j close to i to handle the requests that cannot be served on i.

NEPTUNE ensures minimal service interruption when new placement actions are computed. Existing functions are deleted only after serving the requests they were handling (within a maximum timeout), and nodes are turned off only when all function instances placed on them are either removed or migrated to another node.

Figure 2 details Community 1. Each node hosts a set of instances of the three functions introduced before (□, ○, ◇) and defines proper routing policies. For example, Node N1 runs an instance of □ and ○. Its routing policies state that 100% of the workload for function ◇ must be forwarded to Node N3 since it is the only node in the community that has a running instance of ◇. The requests for functions □ and ○ must be shared, according to the given percentages, with nodes N2 and N3, since they are running other instances of the two functions. Given that Node N1 is equipped with a GPU and function ○ can be accelerated, an additional routing policy defines that within the 50% of requests for ○ that are handled locally, 95% of them are executed on the GPU while 5% on CPUs. Percentages are recomputed every time the Community level is re-executed.

Unlike the first two control levels that mainly target network delay (D), the Node level is dedicated to allocating CPU cores to function instances on the nodes in charge of processing them. In particular, it aims to optimize the sum of queue Q and execution E times, that is, the *handling time* or QE. Each function instance is equipped with a dedicated ***Proportional Integral*** (**PI**) controller that continuously provides vertical scaling of CPU cores. This means that function instances are reconfigured on the fly without the need of restarting them. This way, service disruption, which usually occurs in other solutions [2, 45], is avoided.

The PI controller is fed periodically with monitoring data on the performance of the controlled function instance. In turn, it continuously computes the optimal CPU core allocation to make the handling time meet a user-defined set point. These controllers are independent of each other, and they are not aware of the resource allocations computed by the others. This means that the sum of the resource allocations computed by the PI controllers of the different function instances that

are running on a node can exceed the capacity of the node itself. To avoid resource contention a dedicated component, called *Contention Manager*, gathers proposed CPU allocations and, if needed, scales them down proportionally to fit available node resources. Figure 2 illustrates the details of Node N1. The CPU cores of each running function instance (□ and ○) are vertically scaled by a dedicated PI controller, and resource contentions are handled by *Contention Manager*.

The three control levels work at different time frequencies but in a coordinated way to avoid potential interference. PI controllers (reasoning) and vertical scaling (actuation) are extremely fast so that the Node level can operate every few seconds and quickly react to spikes in the workload. The Community level calculates function placement and routing policies every few minutes so that the underlying infrastructure is fully exploited. Finally, the control period of the Topology level is generally slower. This level only intervenes when the two other control levels struggle in meeting foreseen response times. In case a node fails or is added to the network, the Topology level is automatically activated to properly find a new efficient network partitioning. The hierarchical architecture allows *NEPTUNE* to *scale* efficiently: it maximizes the independence among control levels and the introduced communication overhead is negligible.

3 PLACEMENT, ROUTING, AND ALLOCATION

After partitioning the topology into independent communities, *NEPTUNE* places function instances to minimize latency and the number of active nodes in the MEC infrastructure. The easiest solution is to replicate each function on each node, but available resources could not be enough to take care of all the replicas and would require all nodes to be active. Efficient placement solutions should avoid saturating resources, to not compromise performance, and should minimize service disruption, that is, the continuous function migration among nodes[4]. A static placement cannot take fluctuating workflows into account.

Placement approaches must also consider *graceful termination periods* and *cold starts*. The former is the time the system must wait before terminating a function instance, to let it complete serving the workload it is already handling. The latter is the time a newly created function is ready to serve requests. Both times could range from seconds to minutes [59]. Usually, functions are deployed in lightweight containers whose execution environment must be initialized before the function instance is fully working. Some solutions have been presented in the literature to reduce cold starts [30, 49], but they target specific function types and are not able to always mitigate them significantly. For this reason, *NEPTUNE* does not employ them but uses a custom algorithm.

To tackle these issues, the Community level uses two 2-step optimization processes based on MIP. The first optimization takes care of allocating GPUs; the second schedules the remaining workload on available CPUs. The first step—of each optimization—computes the most efficient function placement and routing policies to minimize the overall network latency and the number of active nodes required to manage the incoming workload. The second step searches for solutions with a similar quality with respect to the one computed in the first step, but that also minimizes the difference between current and newly computed placement. This is to minimize the number of deployment actions and, consequently, minimize service disruption.

Table 1 summarizes the inputs the user must specify for each function, the infrastructure and performance data that are employed in the optimization process, and the decision variables used in the objective functions. For each variable, we reported its name, the data type (e.g., integer, float) and the range of values it can assume.

[4]Note that *NEPTUNE* does not manage state migration and assumes that functions be stateless. In case of stateful function, *NEPTUNE* assumes the states be stored on external components like databases.

Table 1. Inputs, Data, and Decision Variables

Inputs	Type	Range	Description
k_j	Integer	(0, inf)	Cost of running node j
K	Integer	(0, inf)	Maximum allowed overall cost
m_f^{CPU}	Integer	(0, inf)	CPU memory required by function f
m_f^{GPU}	Integer	(0, inf)	GPU memory required by function f
ϕ_f	Integer	(0, inf)	Maximum allowed network delay for function f
α	Float	[0, 1]	Tradeoff between network delay and cost minimization
ϵ	Float	(0, inf)	Tradeoff between placement optimization and service disruption
Infrastructure data			
M_j^{CPU}	Integer	[0, inf)	Memory available on node j
M_j^{GPU}	Integer	[0, inf)	GPU memory available on node j
U_j^{CPU}	Integer	[0, inf)	CPU cores on node j
U_j^{GPU}	Integer	[0, inf)	GPU cores on node j
Monitored data			
$\delta_{i,j}$	Integer	(0, inf)	Network delay between nodes i and j
O_{best}	Float	[0, inf)	Objective function value found after step 1
$\lambda_{f,i}$	Integer	[0, inf)	Incoming f requests to node i
u_j^{CPU}	Float	[0, inf)	Average CPU cores used by node j per single f request
u_j^{GPU}	Float	[0, inf)	Average GPU cores used by node j per single f request
Decision variables			
$x_{f,i,j}^{CPU}$	Float	[0, 1]	Fraction of f requests sent to CPU instances from node i to j
$c_{f,j}^{CPU}$	Boolean	{0, 1}	1 if a CPU instance of f is deployed on node j, 0 otherwise
y_j^{CPU}	Boolean	{0, 1}	1 if any CPU instance is deployed on node j, 0 otherwise
$x_{f,i,j}^{GPU}$	Float	[0, 1)	Fraction of f requests sent to CPU instances from node i to j
$c_{f,j}^{GPU}$	Boolean	{0, 1}	1 if a GPU instance of f is deployed on node j, 0 otherwise
y_j^{GPU}	Boolean	{0, 1}	1 if any GPU instance is deployed on node j, 0 otherwise
MG_f	Integer	[0, inf)	Number of f migrations
CR_f	Integer	[0, inf)	Number of f creations
DL_f	Integer	[0, inf)	Number of f deletions

3.1 Function Placement

NEPTUNE first allocates functions on nodes equipped with GPUs, and then considers CPUs. The formulation presented below is generalized since the two optimization processes are similar. Some of the employed data are resource-specific (e.g., $x_{f,i,j}$, m_f): while in Table 1 we use *GPU* or *CPU* superscript to differentiate them, the rest of the section omits these superscripts for the sake of simplicity (differences between the two processes, when present, are explicitly stated).

Network delay and cost minimization. The first optimization step aims to place function instances in a community $C \subseteq N$ and generate routing policies that minimize both network latency and the overall cost of running the MEC infrastructure.

This optimization problem uses three decision variables: $x_{f,i,j}$, $c_{f,j}$ and y_j. The first one ($x_{f,i,j} \in [0:1]$) represents the fraction of incoming f requests[5] ($\lambda_{f,i}$) that node i routes to node j (i.e., routing policies). The second one ($c_{f,j}$) is a boolean variable that is set to *true* if an f instance is placed onto node j (i.e., placement), and false otherwise, and the third one (y_j) is also a boolean variable that is set to *true* if any (GPU/CPU) function instance is deployed onto j.

The objective function aims to minimize the sum of two contributions: the total network delay (ND) and the cost of the MEC infrastructure (CO), that is, the sum of the costs of the running nodes. ND is defined in Formula 1 as the total sum of each inter-node delay $\delta_{i,j}$ between node i and j multiplied by the incoming workload in i ($\lambda_{f,i}$) and the fraction of such a workload that is routed from i to j ($x_{f,i,j}$):

$$ND = \sum_{f}^{F} \sum_{i}^{C} \sum_{j}^{C} \boldsymbol{x_{f,i,j}} * \delta_{i,j} * \lambda_{f,i}. \tag{1}$$

If we only used the first two factors ($x_{f,i,j} * \delta_{i,j}$), we would have assumed that the incoming workload was distributed equally among nodes (i.e., each node receives the same amount of requests). For this reason, we introduced the per-node incoming traffic ($\lambda_{f,i}$) as a weight. This way, the higher the workload in a node is, the more important its optimization becomes. Note that the workload $\lambda_{f,i}$ that is to be scheduled on CPUs is recomputed by taking into consideration the requests already assigned to GPUs.

The cost of the infrastructure is computed as the sum of the cost of each active node:

$$CO = \sum_{k}^{C} \boldsymbol{y_j} * k_j. \tag{2}$$

The overall objective function is described by Formula 3. As the two contributions assume different value ranges, a simple sum of the two factors could lead to inappropriate values. To address this issue, we normalize both factors, and scale them between 0 and 1 based on their minimum and maximum values. Furthermore, to let users handle the tradeoff between network delay and cost, *NEPTUNE* provides parameter $\alpha \in [0,1]$ to configure the importance of each contribution (i.e., $\alpha = 1$ minimizes the overall cost of running MEC nodes, $\alpha = 0$ minimizes network delay, and $\alpha = 0.5$ assigns equal weights to both contributions).

$$min \quad (1-\alpha) * \frac{ND}{ND_{max}} + \alpha * \frac{CO}{CO_{max}}. \tag{3}$$

To properly minimize network delay, we must define a set of constraints. First, the workload cannot be routed too far from where it enters the system. A parameter ϕ_f defines the maximum allowed network latency of each f request:

$$if\ (\boldsymbol{x_{f,i,j}} > 0)\ \delta_{i,j} \leq \phi_f \quad \forall i,j \in C, \forall f \in F. \tag{4}$$

Second, if a node is configured to execute requests for a certain function f, either directly received or forwarded by other nodes, then it must run an f instance to serve them:

$$\boldsymbol{c_{f,j}} = if\left(\sum_{i}^{C} \boldsymbol{x_{f,i,j}} > 0\right) 1\ else\ 0 \quad \forall j \in C, \forall f \in F. \tag{5}$$

Third, the sum of the memory requirements of the different functions (m_f) placed on j must be less than the capacity of node M_j. Furthermore, a fraction of the node's memory, equal to the

[5]For simplicity, an f request is a user request for function f, and f instance is an instance of function f.

maximum memory requirements of all functions, is always reserved to ensure that new functions can be created before existing ones are removed (as detailed in Section 3.2).

$$\sum_{f}^{F} \boldsymbol{c_{f,j}} * m_f + \max_{f \in F}(m_f) \leq M_j \quad \forall j \in C. \tag{6}$$

Fourth, the computational power of the GPUs and CPUs of a node must be sufficient to handle the incoming workload so that the sum of the average GPU (CPU) resource utilization ($u_{f,j}$) for each processed f request is lower than or equal to the capacity of the node (U_j):

$$\sum_{i}^{C} \sum_{f}^{F} \boldsymbol{x_{f,i,j}} * \lambda_{f,i} * u_{f,j} \leq U_j \quad \forall j \in C. \tag{7}$$

Fifth, all the requests toward a node i must be either executed locally or forwarded to a node j, which means that the generated routing policies cover all the incoming traffic for each node:

$$\sum_{j}^{C} \boldsymbol{x_{f,i,j}} = 1 \quad \forall i \in C, \forall f \in F, \tag{8}$$

when $i = j$, $x_{f,i,j}$ defines the fraction of requests executed locally on i (using CPUs or GPUs), while when $i \neq j$, we consider remote executions.

Sixth, a node j is active if there is at least one instance (to be) deployed onto j:

$$\boldsymbol{y_j} = if \left(\sum_{f}^{F} \boldsymbol{c_{f,j}} > 0 \right) 1 \; else \; 0 \quad \forall j \in C, \forall f \in F. \tag{9}$$

The final constraint serves to restrict the total cost of the infrastructure. In particular, *NEPTUNE* allows users to specify a maximum budget K to ensure an efficient selection of the nodes to keep active. The constraint is expressed as

$$\sum_{i}^{C} \boldsymbol{y_j} * k_j \leq K \quad \forall j \in C \quad . \tag{10}$$

Note that, after assigning the workload to GPUs, the budget K available for CPUs is recomputed by taking into consideration the budget spent for GPUs. Similarly, when a node i is enabled when considering GPU workloads (i.e., $y_j^{GPU} = 1$), it is also considered to be enabled when dealing with the remaining CPU workload (i.e., $y_j^{CPU} >= y_j^{GPU}$) and its cost k_j is set to be zero (since its actual cost have been already subtracted from K while considering GPUs.

This formulation finds the best placement and routing policies that minimize the overall network latency and the cost of running the MEC infrastructure. However, the optimal solution may require several disruptive deployment actions (i.e., creations, migrations, or deletions). To address this problem, the second step of the optimization process allows for reducing service disruption and improving the result.

Disruption minimization. The second step aims to generate function placements and routing policies that minimize service disruption. This step looks for solutions that are slightly less efficient than the optimal one computed in the first step but that minimize the number of disruptive actions to execute (e.g., function migration). To do that, the second step keeps the constraints defined in Formulae 4- 10 and adds:

$$(1 - \alpha) * \frac{ND}{ND_{max}} + \alpha * \frac{CO}{CO_{max}} \leq O_{best} * (1 + \epsilon). \tag{11}$$

This is to say that the final placement must have a placement quality in the interval $[O_{best}, O_{best} * (1+\epsilon)]$, where O_{best} is the optimal placement produced in the first step (i.e., the result produced by Formula 3), and ϵ is an arbitrarily small value that (slightly) worsens the solution in terms of network overhead and infrastructure cost. Parameter ϵ essentially quantifies how much we are willing to deviate from the optimal solution in terms of network delay and cost to lessen the impact on the system. It represents a tradeoff between system performance and service disruption. For instance, if ϵ is set to 0.05, we are allowing the system to select a solution that is up to 5% worse than the optimal in terms of network delay and cost but potentially involves fewer migration actions.

We also consider the amount of created, migrated, and deleted f instances between two configurations, that is, between the current placement ($c^{old}_{f,i}$) and the one to be actuated ($c_{f,i}$). DL_f and CR_f define the number of deleted and created instances, respectively, computed as the maximum between 0 and the differences between the instances in the two configurations:

$$\begin{aligned} \boldsymbol{DL_f} &= \sum_{i}^{C} max\left(c^{old}_{f,i} - \boldsymbol{c_{f,i}}, 0\right) \quad \forall f \in F \\ \boldsymbol{CR_f} &= \sum_{i}^{C} max\left(\boldsymbol{c_{f,i}} - c^{old}_{f,i}, 0\right) \quad \forall f \in F \end{aligned} \quad . \tag{12}$$

The number of migrated instances (in the new placement), that is, the amount of function instances moved from one node to another, is defined as the minimum between created CR_f and deleted DL_f:

$$\boldsymbol{MG_f} = min(\boldsymbol{CR_f}, \boldsymbol{DL_f}) \quad \forall f \in F, \tag{13}$$

The new objective function is then defined as

$$min \quad \sum_{f}^{F} \boldsymbol{MG_f} + \frac{1}{\boldsymbol{DL_f} + 2} - \frac{1}{\boldsymbol{CR_f} + 2} \tag{14}$$

The goal of the objective function is to minimize the number of migrations (MG_f). Creations and deletions are needed, to avoid under- and over-provisioning. The two factors $\frac{1}{DL_f+2}$ and $\frac{1}{CR_f+2}$, which are always lower than 1, allow for selecting the best solutions among the ones with the same number of migrations but with a different number of deletions and creations.

The combined effect of the new objective function and the constraint defined in Formula 11 guarantees the generation of placements and routing policies with close-to-optimal placement and with the minimum amount of disruptive actions. The control-theoretical planners at Node level are then in charge of vertically scaling containers as needed.

3.2 State Transition Management

When a new function placement is computed, *NEPTUNE* must handle the transition from the existing placement to the new one. This transition, if not managed correctly, can lead to service unavailability (e.g., a node is shut down before the incoming requests are processed or routed elsewhere). *NEPTUNE* includes a transition algorithm that minimizes service interruption and cold starts. Specifically, the algorithm focuses on (i) removing a function instance only when it served all incoming requests, (ii) ensuring there is always at least a function instance that is serving requests, and (iii) shutting down nodes when no function instances are deployed onto them.

Algorithm 1 describes procedure *StateTransition* employed by *NEPTUNE* for managing state transitions. The algorithm assumes that nodes always be provided with enough resources to create at least a new function instance. This is guaranteed by Equation (6), which ensures that each node maintains sufficient free memory to accommodate an instance of the most resource-intensive

ALGORITHM 1: State Transition Algorithm

```
1:  procedure StateTransition()
2:      Compute nodes to be activated (NodeAct) and nodes to be deactivated (NodeDeact) from {y_j^old} and {y_j}.
3:      Compute delete set (DltSet), create set (CrtSet), and migrate set (MgrSet) by comparing {c_{f,j}^old} and {c_{f,j}}.
4:      ActivateNodes(NodeAct)
5:      for targetNode, functionType in DltSet do
6:          async SafeDelete(targetNode, functionType)
7:      end for
8:      for sourceNode, targetNode, functionType in MgrSet do
9:          async
10:             SafeCreate(targetNode, functionType)
11:             SafeDelete(sourceNode, functionType)
12:         end async
13:     end for
14:     WaitForMigrationsToComplete()
15:     for targetNode, functionType in CrtSet do
16:         async SafeCreate(targetNode, functionType)
17:     end for
18:     WaitForDeletionsToComplete()
19:     DeactivateNodes(NodeAct)
20: end procedure
21: procedure SafeDelete(targetNode, functionType)
22:     RemoveFromRoutingPolicies(targetNode, functionType)
23:     GracefullyTerminate(targetNode, functionType)
24: end procedure
25: procedure SafeCreate(targetNode, functionType)
26:     CreateInstance(targetNode, functionType)
27:     SetupInstance(targetNode, functionType)
28:     AddToRoutingPolicies(targetNode, functionType)
29: end procedure
```

function. First, the algorithm computes the nodes that must be activated or deactivated by comparing the previously activated nodes $\{y_j^{old}\}$ with the newly computed ones $\{y_j\}$ (line 2). At line 3, function instances that need to be managed are split into three different sets: delete set (*DltSet*), create set (*CrtSet*), and migrate set (*MgrSet*). The first two sets include ($targetNode$, $functionType$) pairs that identify the node where a given function type must be deleted/created. The third set includes ($sourceNode$, $targetNode$, $functionType$) triples that define the function type that must be migrated from a source node to a target node.

From an availability perspective, functions in the delete and create sets are less critical compared to those in the migration set. Creations can produce unavailability only when cold starts are not properly managed. Deletion of a function instance may affect requests that are currently processed by the system. However, as a consequence of Equation (8), there must always be at least one instance in a community for each function type. This means that there cannot be a set of deletion actions that remove all the instances for a given function type. Conversely, migrations may induce more severe temporary service unavailability when all existing instances of a function are required to move to different nodes.

Next, the algorithm synchronously activates the nodes that should host new function instances, and that were not activated in the old placement (line 4). Then, for each pair in *DltSet*, *NEPTUNE* calls *asynchronously* (i.e, without waiting for its termination) procedure *SafeDelete* (lines 5–7) that is reported at lines 21–24. In this procedure, the routing policies for a given target node and a function type are removed from all the other nodes so that the function instance ceases to accept

new requests (line 22). Moreover, before terminating the function instance, *NEPTUNE* employs a grace period to serve remaining requests (line 23).

Similarly, in lines 8–13, migrations are handled by calling, asynchronously, procedures *SafeCreate* (on the function type and target node) and *SafeDelete* (on the function type and source node). Procedure *SafeCreate* is reported in lines 25–29. For each function to create, the procedure creates an instance, configures the runtime and downloads needed libraries, and finally allows for incoming traffic by setting up the routing policies. This way, cold starts are mitigated since requests are forwarded to newly created instances only when they are ready to serve requests.

Given that migrations necessitate the creation and subsequent deletion of function instances, the algorithm must wait for these migrations to complete (line 14) before initiating the creation of additional function instances (lines 15–17). This sequence ensures ordered processing and maintains nodes with enough space to host new function instances. Finally, after waiting (if needed) for the completion of delete actions (line 18), the nodes that do not host any instance are deactivated (line 19).

3.3 CPU Allocation

The Node level aims to control QE, that is, the handling time, defined as the sum of the queue time Q and execution time E; while network latency D is already optimized by the Community level. QE is subject to multiple factors, such as fluctuating workloads or changes in the execution environment. The feedback loop in our controllers allows them to handle these changes at runtime by continuously changing the CPU cores allocated to function instances. This way, Node level controllers help meet set response times given a placement produced by the Community level. If the Node level cannot keep QE under control, it means that the nodes are over saturated and the Community level is triggered to compute a new placement and routing policies.

The Node level controllers borrow from our previous work on control-theoretical solutions for dynamic resource management [6] and from other similar works [16]. They exploit lightweight PI controllers to quickly (re-)allocate CPU cores to each function. They feature a fast control period because they can compute the next state of the system in constant time, and they also provide a-priori formal guarantees.

Each function instance is controlled by a dedicated PI controller. The feedback loop monitors the average value of QE, calculates the new core allocation, and actuates it by reconfiguring the container that wraps the function instance on the fly. More formally, given a set point $QE_{f,desired}$ (the desired handling time), the controller periodically reads the current value of $QE_{f,j}$ (controlled variable) from the monitoring infrastructure, that is, the value of QE_f measured on node j, and computes the difference between desired and actual values. Note that, since the controllers aim to keep $QE_{f,j}$ as close to $QE_{f,desired}$ as possible, the set point should be set to a lower value than RT_f^R, the required response time for function f.

PI controllers are reactive. They compute the mismatch between the set point and the monitored value and compute the number of cores that the function should use to minimize the error. Algorithm 2 describes the logic of PI controllers.

Line 2 starts the computation by calculating the error, denoted as err, which is the difference between the inverse of $QE_{f,desired}$ and $QE_{f,j}$. This error serves as a measure of how far the current performance is from the desired one. Line 3 retrieves the current core allocation (*cpu*) for the function instance.

Line 4 calculates the integral contribution from the previous control step, int^{old} as the difference between the current core allocation, *cpu*, the integral gain, g_{int} (a tuning parameter), and the previous error err^{old}. Line 5 computes the new integral contribution, *int*, by multiplying the current

ALGORITHM 2: CPU core allocation at Node level.

1: **procedure** ComputeInstanceCores(f, j)
2: $err := \frac{1}{QE_{f,desired}} - \frac{1}{QE_{f,j}}$;
3: $cpu := getCPUAllocation(f, j)$;
4: $int^{old} := cpu - g_{int} * err^{old}$;
5: $int := int^{old} + g_{int} * err$;
6: $prop := err * g_{prop}$;
7: $cpu_{f,j} := int + prop$;
8: $cpu_{f,j} := max(cpu_j^{min}, min(cpu_j^{max}, cpu_{f,j}))$;
9: $err^{old} := err$;
10: **end procedure**

error, err, and the integral gain, g_{int}, and adding the previous integral contribution, int^{old}. The integral contribution captures the accumulated past errors, and allows for a correction based on the system's history. Then, line 6 calculates the proportional contribution, $prop$, by using the current error, err, and the proportional gain, g_{prop} (tuning parameter). The proportional contribution offers an immediate response to the current error to rapidly bring the system performance closer to the desired state.

Next, line 7 derives the core allocation for function instance f deployed on node j ($cpu_{f,j}$). This allocation is the sum of the integral and proportional contributions, adjusted to fall within the bounds of the minimum and maximum allowed core allocations, cpu_j^{min} and cpu_j^{max}, respectively (line 8). With this new allocation, the system gears up for the next iteration, set to correct the error and maintain the desired set point. Finally, the previous error value, err^{old}, is updated with the current error at line 9 to ensure that the algorithm reflects the most recent state of the system.

The PI controllers dedicated to function instances that share the same underlying node j are independent of the others. Therefore, the computed allocations $cpu_{f,j}$ are not immediately enacted since they could outreach the node's capacity. Thus, the allocations of the different f instances that run on node j are processed by the *Contention Manager* (one per node), which calculates feasible core allocations $cpu_{f,j}^{feas}$. If the sum of $cpu_{f,j}$ fits the node's capacity, they are actuated with no changes. Otherwise, the *Contention Manager* scales them down according to a customizable heuristic. By default, the allocations are scaled-down proportionally, as follows:

$$cpu_{f,j}^{feas} = \frac{cpu_{f,j} * cpu_j^{max}}{max(\sum_f cpu_{f,j}, cpu_j^{max})}. \tag{15}$$

4 PROTOTYPE

Our prototype is built on top of K3s, a lightweight distribution of Kubernetes optimized for edge computing, and is designed to be compatible with OpenFaaS function specifications, a well-known open-source serverless platform.

4.1 Kubernetes/K3S

K3S (Kubernetes) adopts a distributed architecture with a master node (*K3s-server*) that manages a set of workers (*K3s-agents*) in charge of running containers. The master node can be replicated to increase availability; all nodes use a daemon called *kubelet* to ensure that containers be up and running.

Kubernetes uses *resources* to represent both the underlying infrastructure and applications. Some resources are built-in (e.g., pods, nodes, services), but additional ones can be defined by users

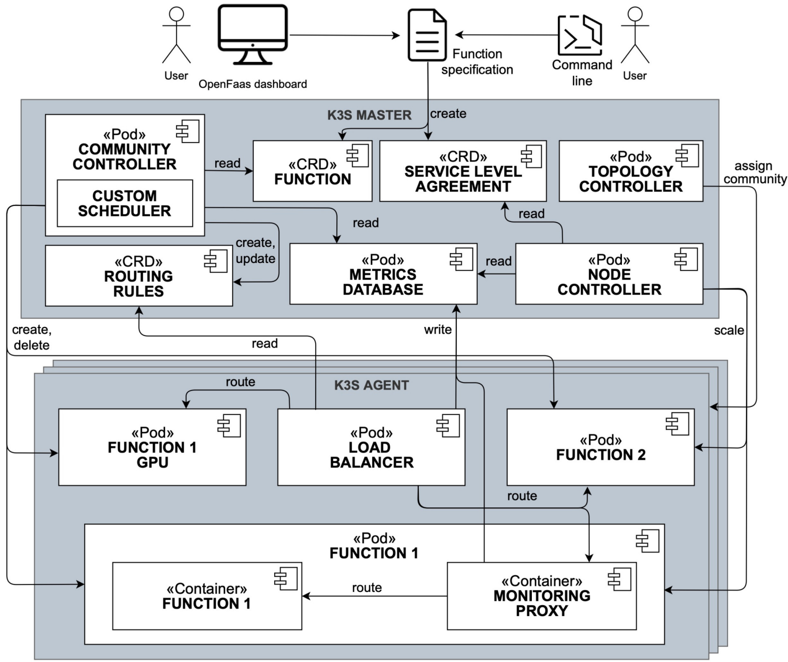


Fig. 3. Component diagram of *NEPTUNE*'s prototype.

by using ***Custom Resource Definitions*** (**CRDs**) (e.g., ad-hoc monitoring, custom routing policies). All resources (built-in and custom ones) are characterized by a *current state* and a *desired state*.

Kubernetes manages resources by using the pattern *controller*. Users declare their goals (e.g., the desired response time) by configuring the desired state of resources, and Kubernetes employs *controllers*, which are components that implement control loops, to make the current state of resources move closer to their desired state. One can deploy custom controllers as well as manage CRDs created by users. For example, a *pod*, which is the smallest resource that can be deployed in the infrastructure and wraps one or more containers into a single isolated environment, can be configured with a minimum and maximum amount of computing resources assigned to it. Then, a built-in controller named *kube-scheduler* periodically checks for unscheduled pods and assigns them to nodes to let them have the amount of resources specified in the configuration.

To help controllers achieve their goals, Kubernetes provides a monitoring system, called *metric-server*. For instance, the kube-scheduler employs a scoring algorithm that consumes data from metric-server (e.g., CPU or available memory) to optimize the placement of pods. The monitoring system and the scheduler can be replaced by custom components to accommodate a larger variety of metrics (e.g., response time, network delay) and scheduling strategies (e.g., *NEPTUNE*'s function placement). Hardware heterogeneity, and thus GPUs, is managed by a dedicated built-in component called *device plugin*. This plugin assigns a GPU to a pod at a time, which holds the ownership for its whole lifetime.

4.2 Architecture

Figure 3 shows the architecture of our prototype[6]. The prototype assumes that each node corresponds to a K3s-agent running in the cluster. Each control level is embedded in a dedicated

[6]Available at https://github.com/deib-polimi/neptune

component that implements it. In particular, component *Topology controller* dynamically splits the network using the SLPA algorithm, component *Community controller* is in charge of generating the function placement and routing policies for each community, and component *Node controller* embeds all the PI controllers that perform dynamic CPU provisioning at Node level.

Since existing monitoring solutions do not provide means to monitor and collect the fine-grained metrics required by *NEPTUNE*, we implemented a custom monitoring system that exploits the *Sidecar pattern*[7]. This pattern aims to create two or more containers in the same pod where one embeds the application logic, and the others perform non-functional tasks such as monitoring, logging, and configuration management without requiring modifications to the running application. Thus, the prototype employs a custom lightweight proxy (as sidecar) that is automatically injected inside all function pods, intercepts requests, and collects required metrics. Monitoring proxies push collected data into component *Metrics Database*, which is used by the controllers to be informed of the performance of the system.

The Topology controller periodically checks for changes in the node network (e.g., when a node is added or removed) and executes the SLPA algorithm to reorganize nodes into communities. The algorithm is configured using a CRD that allows one to specify the maximum size of a community, the maximum delay between the nodes in a community, and the maximum number of iterations of the SLPA algorithm.

The functions can be deployed using the CRD *Function* provided by OpenFaaS. The user must provide the function name, the container image to use, the maximum network delay, if it can exploit GPUs, and the amount of GPU memory it needs. If not specified, GPU acceleration is disabled by default.

NEPTUNE provides the CRD *ServiceLevelAgrement* to set the desired response time RT_f^R of the function. This resource allows one to specify the minimum and maximum amount of resources to be assigned to function instances, and also which container (of a pod) the Node level should manage when function pods are composed of multiple containers.

Once the CRDs *Function* and *ServiceLevelAgrement* are created, the Community controller addresses the new function in the placement process. Since communities are independent, at least one instance of the (new) function will be created in each community. The placement takes into account the functions that require GPUs and creates pods able to exploit the dedicated hardware. Whenever a new placement is computed, *NEPTUNE* routing policies are written in an additional CRD named *CommunitySchedule*, which is then consumed by the load balancers. In particular, the CRD *CommunitySchedule* is automatically managed by *NEPTUNE*, and no manual intervention is needed.

State-of-the-art load balancers are not designed to handle frequent changes in routing policies and they must be restarted every time the internal configuration is modified. For this reason, we developed a custom load balancer that is able to periodically change its routing policies without service interruption. The prototype deploys an instance of this load balancer on each node. When the requests for a deployed function f reaches a node j, the load balancer (running on j) collects the arrival rate $\lambda_{f,j}$ and the average response time $rt_{f,j}$, and stores these data into the Metrics Database. With these data, the Community controller may (i) move the instances closer to the traffic's source, (ii) adjust function instances according to the workload, and (iii) properly update routing policies by using the formulation described in Section 3.

The heuristic used by the kube-scheduler to deploy containers could interfere with the logic of the Community controllers. For this reason, our prototype uses a custom scheduler, implemented

[7] https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar

as a sub-component of Community controller, which directly enacts the newly computed placements. The prototype supports a seamless evolution of the system. Old instances are not deleted until the new ones are ready to serve requests. Furthermore, the deletion occurs using graceful termination periods, giving time to all instances to serve pending requests. Then, load balancers update their routing policies to guarantee service availability: whenever new routing policies are rolled out, the prototype waits until the new pods are available before replacing the old ones.

Each Node controller is in charge of scaling the CPU cores associated with each function instance (vertical scaling). The current standard distribution of Kubernetes does not allow one to change allocated resources without restarting pods, a process that sometimes can take minutes. This could compromise the Node level, which would become incapable of handling bursty workloads. As a result, the prototype augments Kubernetes with the ***Kubernetes Enhancement Proposal*** (**KEP**) 1287 that implements *In-Place Pod Vertical Scaling*[8] and allows pods' resources to be changed without restart. This enables Node controllers to operate at a faster pace.

To achieve coordination and scalability, our prototype minimizes communication overhead and dependencies among components and controllers. Each controller continuously monitors the desired infrastructure state described in the CRDs and reconciles it with the current state independently. When information (e.g., response time) needs to be shared across different controllers, data are written in a shared storage (i.e., Metrics Database) and, in turn, are read asynchronously by other components.

The prototype uses *nvidia-docker*, a container runtime that enables the usage of GPUs inside containers. However, by default, GPU access can only be reserved for one function instance at a time. This prevents the full exploitation of GPUs and limits the possible placements produced by Community controllers. To solve this problem, the prototype employs a device plugin [9] developed by AWS instead of the default one. This plugin abstracts a GPU by creating multiple *Virtual GPUs* (vGPUs), each one reserved to a function instance, and then it makes use of the *Nvidia Multi-Process Service*[10] (MPS), a runtime infrastructure designed to transparently allow GPUs to be shared among multiple processes (e.g., containers), to enable the fractional allocation of GPUs.

When available GPUs are not enough to carry out all requested computations, we create an additional container for the same function, which only exploits CPUs (as Function 1 in Figure 3). This replication is mandatory since the runtime solutions dedicated to GPU processing (e.g., TensorFlow serving[11]) do not allow the same container to exploit both GPUs and CPUs.

In total, the prototype comprises three custom resources and six controllers for optimal function placement, smart routing policies, request forwarding, and monitoring. The prototype requires the installation of 2 external components (i.e., AWS Plugin and Nvidia Multi-Process Service) for shared and transparent access to GPUs, and it needs to enhance K3s with KEP 1287 to enable restart-free vertical scaling.

5 EVALUATION

The solution adopted at Topology level is largely covered by PAPS [5]. The experiments in this article focus on evaluating Community and Node level. The conducted evaluation aims to both evaluate the single aspects and provide an assessment of the overall system by addressing the following research questions:

[8] https://github.com/kubernetes/enhancements/tree/master/keps/sig-node/1287-in-place-update-pod-resources
[9] https://github.com/awslabs/aws-virtual-gpu-device-plugin
[10] https://docs.nvidia.com/deploy/mps/index.html
[11] https://www.tensorflow.org/tfx/guide/serving

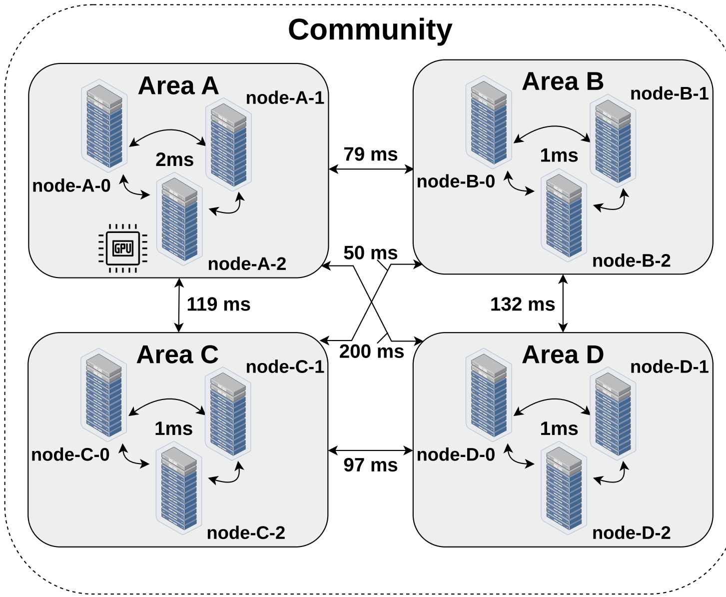


Fig. 4. Network delays between inter- and intra-area nodes.

RQ1 How can *NEPTUNE* scale provisioned resources automatically?
RQ2 How effective is *NEPTUNE* in managing resource contentions on nodes?
RQ3 How efficient can *NEPTUNE*'s GPU management be?
RQ4 How well does the placement and routing strategy of *NEPTUNE* handle real-world workloads?

5.1 Experimental Setup

Infrastructure. To emulate a MEC Topology, we employed a geo-distributed cluster of AWS EC2 virtual machines. We deployed the virtual machines using both different regions and availability zones to have significant network delays that resemble the ones in an edge environment. We used four different areas, each corresponding to an AWS region: Area A to *eu-west (Ireland)*, Area B to *us-east (Ohio)*, Area C to *us-west (Oregon)* and Area D to *ap-northeast (Tokyo)*. Within the same area, the nodes were deployed onto different AWS availability zones to avoid close-to-zero network delays. The average network delays between each pair of areas are computed as the round trip times of an ICMP [40] (Internet Control Message Protocol) packet. Obtained results are shown in Figure 4. The average inter-area delays between nodes in two different regions range from 50ms and 200ms, while intra-area delays between nodes within different availability zones are between 1ms and 2ms.

To assess the performance of *NEPTUNE* we used a large variety of nodes. As master node, we used a *c5.2xlarge* instance (8 vCPUs, 16 GB memory), while as worker nodes we employed instances of different virtual machines. To represent MEC nodes with few computing resources, we deployed the workers on *m5.xlarge* instances (4 vCPUs, 16 GB memory) and *c5.xlarge* instances (4 vCPUs, 8 GB memory). Nodes with many resources were represented by *c5.9xlarge* instances (36 vcpus, 72 GB memory), and we employed *g4dn.xlarge* instances (4 vcpus, 16 GB memory, 1 GPU) as GPU-accelerated nodes.

NEPTUNE settings. The Topology controller was configured to run every 10 minutes and when the cluster was subject to a topology change. The Community controllers recompute the function

Table 2. Characteristics of Deployed Functions

Name	Language	Memory	RT	QE	Cold Start
SeBS benchmark					
compression	Python	~1GB	200ms	100ms	5s
dynamic-html	Python	~1GB	200ms	100ms	5s
graph-bfs	Python	~1GB	200ms	100ms	5s
graph-mst	Python	~1GB	200ms	100ms	5s
thumbnailer	Python	~1GB	200ms	100ms	5s
video-processing	Python	~4GB	20s	10s	5s
Machine Learning					
resnet	Python	~500MB	550ms	275ms	100s

placement and the routing policies every minute. Node controllers were configured with a control period of 5 seconds. One can also use faster control loops, but they may lead to inconsistent resource allocation updates since K3S resource states are stored in a remote database. The integral gain g_{int} of PI controllers was set to 25 and the proportional gain g_{prop} to 50 in almost all cases and when different values were used, we report them explicitly.

Applications. In our previous work [4], we evaluated *NEPTUNE* using an application (*sock-shop*[12]) composed of multiple functions that are dependent on one another. In this article, we conducted experiments with a new set of functions as reported in Table 2. We borrowed the functions from the literature [12, 42]. Overall, the set of employed functions is characterized by multiple memory requirements (ranging from 500MB to 4GB) and cold start times (from a few seconds to minutes).

The first benchmark is *SeBS* [12], a set of independent serverless functions widely employed in the literature. As shown in Table 2, we selected some of the functions of the benchmark, and we made small changes to make them compatible with the *OpenFaas* function interface (e.g., the addition of health check endpoints). The second application is *resnet* [56], a machine learning inference application. Resnet was deployed on top of Tensorflow Serving and can run using either CPUs or GPUs.

Collected metrics. We employed multiple metrics to evaluate the performance of *NEPTUNE*. In particular, we collected (i) *response time* (ms) to measure the end-to-end delay for an edge-client to send a request and receive the corresponding response, (ii) *response time violation rate* (% of requests) to measure the percentage of requests that were not served within the required response time RT_f^R, (iii) *network time rate* (% of response time) to quantify the amount of time spent by a request in the network (e.g., being forwarded from one node to another) with respect to the amount of time taken to process the request, (iv) *allocated cores* (millicores or thousandths of a core) to quantify the amount of resources allocated to each function.

To present consistent and statistically relevant results, each experiment was run 5 times. We use μ to denote observed average performance and σ to represent standard deviation.

Workload shapes. We have designed multiple workloads to accurately simulate various user behaviors, encompassing both synthetic scenarios and real-world traces and datasets. The first synthetic workload, denoted as *Ramp(MinUsers, MaxUsers, StartTime, RampTime, Duration)*, captures the gradual increase in user requests over time. Starting from an initial *MinUsers* number of users (concurrent requests), the workload remains constant for *StartTime* seconds. Subsequently, over

[12] https://github.com/microservices-demo/microservices-demo

a span of *RampTime* seconds, the workload steadily ramps up to reach the maximum number of users, *MaxUsers*. Following this increase, the number of users remains consistent for the remaining *Duration* of the experiment to allow us to thoroughly assess system performance at steady state.

The second synthetic workload we employed, named *Syn(MinUsers, MaxUsers, StartTime, Period, Duration)*, replicates user behaviors through a sinusoidal pattern, and creates a dynamic fluctuation of user counts between *MinUsers* and *MaxUsers*. Initially, the workload begins with a period with a constant number (*MinUsers*) of users, followed by a sinusoidal shape that extends for the entire *Duration* of the experiment. Parameter *Period* determines the duration of a complete sinusoidal cycle, and allows us to evaluate the system's robustness under varying user loads.

In addition to synthetic workloads, we included traces[13] generated from three distinct real-world datasets. The first workload W_{cs} is generated from *Cabspotting* [39], a dataset containing GPS traces from 500 taxi drivers operating in San Francisco over one month. The second workload W_{td} is generated from *T-Drive* [62], a dataset featuring more than 10,000 taxi drivers in Beijing over a timeframe of one week. The third workload W_{tel} is generated from a dataset called *Telecom* dataset [18], sourced from Shanghai Telecom, which contains records of internet access through more than 3,000 base stations by almost 10,000 mobile phones, over a six month period.

To adapt these real-world traces to our MEC topology, we undertook three pre-processing steps. First, we normalized the geographic positions of the users, spreading them across the entire globe to align with our execution environment. This adjustment ensures that our evaluation captures a representative distribution of user locations and their associated resource demands. Second, we introduced a think time for each user to simulate realistic user behavior and workload fluctuations. The think time accounts for user interactions, delays, and idle periods, and aligns the workload characteristics to real-world scenarios. Third, selected traces were originally collected over different timeframes (from one week to six months). For this reason, we normalized the timestamps of all traces such that the workload generated from these traces was characterized by the same duration.

Competitors. We compared *NEPTUNE* against different well-known approaches. To evaluate the autoscaling capability, we confronted *NEPTUNE* against two autoscaling solutions: *Horizontal Pod Autoscaler*[14] (HPA) and *Vertical Pod Autoscaler*[15] (VPA). HPA provides horizontal autoscaling and employs threshold rules to actuate resource provisioning. VPA also adopts threshold rules, but it scales function instances (vertically). Note that VPA does not offer an in-place vertical scaling solution (as the one provided by *NEPTUNE*): it requires that function instances be restarted to change allocated resources.

Additionally, we compared *NEPTUNE* against three different approaches for placement and routing, namely VSVBP [26] (Variable Sized Vector Bin Packing), MCF [27] (Most Capacity First) and CR-EUA [34] (Criticality-Awareness Edge User Allocation). The first two (VSVBP and MCF) are popular approaches often used in the literature as baseline for edge resource allocation solutions [20, 33, 55], while CR-EUA is a recent extension of the previous approaches for safety-critical, low-latency applications. Specifically, VSVBP maximizes the number of allocated requests and minimizes the number of active edge nodes while ensuring the response time of the deployed services. MCF models the problem in a similar way to VSVBP, but it handles dynamic response time requirements and it proposes a faster but sub-optimal algorithm. CR-EUA proposes a solution for safety-critical applications that aims to maximize the number of requests processed with the highest level of criticality. We implemented competitors' approaches in the Community level by

[13]The workload generator is available at https://github.com/deib-polimi/neptune-workload
[14]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale
[15]https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler

Table 3. Evaluation of Autoscaling Capabilities and Comparison Against Competitors

Function		Response Time (ms) ↓			Response Time Violations (%) ↓			Core Allocations (millicores) ↓		
		NEPT	HPA	VPA	NEPT	HPA	VPA	NEPT	HPA	VPA
compression	μ	79.5	82.6	**51.3**	3.8	7.3	**0.3**	**1239.5**	1731.1	1820.5
	σ	1.1	16.4	1.4	0.3	2.6	0.2	48.7	169.8	32.5
dynamic-html	μ	94.1	**80.6**	199.0	**4.4**	4.6	38.7	**1814.6**	2281.9	2947.2
	σ	0.8	5.3	7.4	0.6	1.0	1.7	69.6	69.6	30.4
graph-bfs	μ	**105.6**	107.6	8981.4	**1.0**	1.1	100.0	2620.1	11043.2	**355.8**
	σ	1.6	2.4	5640.7	0.2	0.6	0.0	837.4	6317.6	110.4
graph-mst	μ	99.5	**84.7**	4753.8	2.1	**1.2**	99.8	1392.0	3321.5	**406.3**
	σ	1.0	5.0	3319.6	0.7	0.3	0.3	102.2	1539.7	182.6
thumbnailer	μ	98.9	**89.3**	5778.3	3.6	**2.3**	99.0	2034.4	3381.3	**420.6**
	σ	0.6	1.4	3838.6	0.6	0.5	1.7	33.4	560.3	195.3
video-processing	μ	**10151.9**	15500.0	41269.7	**0.0**	14.6	89.0	12053.2	18137.0	**2291.7**
	σ	9.5	1741.4	8328.4	0.0	9.8	10.2	202.9	2100.9	958.6

Best results are highlighted in bold.

replacing our formulation. To provide a fair comparison, the in-place vertical autoscaling feature, provided by *NEPTUNE* at the Node level, was enabled for all the competitors.

5.2 RQ1: Autoscaling Capabilities

The first set of experiments aimed to evaluate how *NEPTUNE* allocates resources with respect to popular autoscaling approaches (i.e., HPA and VPA). These solutions can control duration *QE* by increasing or decreasing the amount of computing resources allocated to a function. Since this set of experiments only focuses on evaluating the effectiveness of resource allocation, we decided to use an infrastructure with a single worker node to reduce the measurement noise that network delays might introduce. The master node was deployed on top of a *c5.2xlarge* virtual machine in *Area A*, while the worker node used a *c5.9xlarge* virtual machine. These experiments employed the functions provided in SeBS [12]. The benchmark covers several functions: from ones used for multimedia and utilities to the ones for web apps and scientific computations. We configured the functions as reported in Table 2. We employed a sinusoidal workflow configured with $MinUsers = 1$, $MaxUsers = 50$, $StartTime = 60s$, $Period = 1200s$, and $Duration = 1200s$.

Table 3 shows obtained results. We can observe that *NEPTUNE*, in most cases, was able to allocate enough resources to achieve the desired response time. The average and median response times were close to *QE* and well below RT^R, the desired response time. For example, if we consider functions *compression* and *graph-bfs*, they achieved an average response time equal to 79.5*ms* and 105.6*ms*, respectively. The response time violation rates were also really low: 3.7% for *compression* and less than 0.9% for *graph-bfs*. Similar considerations can be made for three other functions with the same response time requirements: *dynamic-html*, *graph-mst*, and *thumbnailer*.

The table also shows the results obtained by using HPA and VPA. Among the three approaches, VPA was the worst. Due to design limitations in Kubernetes (illustrated in Section 5.1), VPA requires that pods be restarted to change allocated resources. This negatively impacts the reactiveness of the approach and worsens the effectiveness of autoscaling capabilities. The results in the table show that VPA achieved reasonable response times only for function *compression*: an average response of 51.3*ms* and a response time violation rate lower than 0.3%. As for the other functions, VPA failed to ensure the desired response time and most of the requests ended in response violations. For example, functions *graph-bst*, *graph-mst* and *thumbnailer* were characterized by a response time violation rate equal to 100%, 99.8% and 99.0%, respectively.

Table 4. Results of Autoscaling Experiments with Different Gains (i.e., $g_{int} = k \cdot 25$ and $g_{prop} = k \cdot 50$)

Function		Response Time (ms) ↓			Response Time Violations (%) ↓			Core Allocations (millicores) ↓		
		k = 10^0	k = 10^1	k = 10^2	k = 10^0	k = 10^1	k = 10^2	k = 10^0	k = 10^1	k = 10^2
video-processing	μ	26167.4	18983.8	11651.1	57.1	45.6	0.0	**3651.6**	5110.6	8896.8
	σ	41.1	26.6	9.5	0.3	0.5	0.0	17.4	22.1	36.4
		k = 10^3	k = 10^4	k = 10^5	k = 10^3	k = 10^4	k = 10^5	k = 10^3	k = 10^4	k = 10^5
video-processing	μ	**10151.9**	10894.8	12741.2	**0.0**	2.2	12.7	12053.2	17523.4	18273.7
	σ	9.5	256.2	416.9	0.0	0.8	1.4	202.9	636.8	694.6

Best results are highlighted in bold.

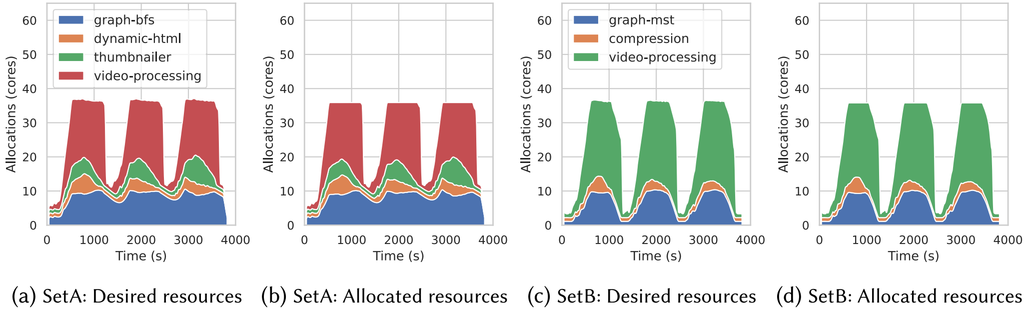
On the other hand, HPA was more effective in ensuring response times than VPA. For instance, the average response times for function *compression* and *dynamic-html* were 82.6ms an 80.6ms, with response time violation rates equal to 7.3% and 4.6%, respectively. Autoscaling was performed similarly also for the other three functions with equal desired response time (i.e., *graph-bfs*, *graph-mst*, *thumbnailer*). Compared with *NEPTUNE*, HPA achieved slightly worse performance when dealing with these functions. HPA was able to ensure a slightly lower average response time than *NEPTUNE*: short-running functions achieved an average response time of 88.9ms, and of 95.5ms with *NEPTUNE*. HPA reported higher variance (i.e., the average standard deviation for short-running functions was 6.1ms for HPA and 1.0ms for *NEPTUNE*). Then *NEPTUNE* was able to guarantee more stable response times over time. In fact, despite a higher average response time, the violation rate was equal to 3.0% with *NEPTUNE* and to 3.3% with HPA. This behaviour is mainly due to the fact that HPA employs a horizontal scaling approach, which means that it can either create a new replica or delete an existing one. Since each replica has a certain amount of resources assigned to it, HPA can allocate (or deallocate) a fixed amount of resources (usually in the order hundreds of millicores), while *NEPTUNE* can perform a finer-grained resource allocation (in the order of single millicores). For this reason, when it comes to allocating the right amount of resources, HPA is less precise than *NEPTUNE*. The observation on resource allocation further confirms the suboptimality of HPA. HPA allocated 4.35 cores on average while *NEPTUNE* only allocated 1.82 cores (59.2% fewer allocated resources).

The last row of the table refers to function *video-processing*. This function is characterized by being a long-running function since, unlike the other functions that take a few hundred milliseconds, it requires some seconds to compute the response. To cope with the higher response time, we employed higher integral and proportional gains (i.e., $g_{int} = 25000$ and $g_{prop} = 50000$). The average response time achieved by *NEPTUNE* was 10.1s, which was very close to the target one (i.e., 10s), and all requests were served within the desired response time RT^R.

Table 4 shows how the performance of function *video-processing* changed when we used different integral and proportional gains. Note that when the gains of the PI controllers were too low (i.e., k = 10^0 and k = 10^1), the controllers could not allocate enough resources to cope with the workload. The average response times were 26.2s and 18.9s, in the two cases, and about half of the requests exceeded the desired response time.

On the other hand, when the gains were very high (i.e., k = 10^4 and k = 10^5), the amount of allocated resources was unstable. In fact, despite having acceptable average response times (i.e., 10.9s and 12.7s, respectively), the standard deviations of these runs were higher than the other configurations. The instability led to higher response time violations (i.e., 2.2% and 12.7% of the requests were not served within RT^R) and resource over-provisioning (i.e., 17.5 and 18.3 cores, in the two cases).

The experiments showed that *NEPTUNE* can achieve very good performance if the integral and proportional gains were correctly set, and they highlighted the importance of proper



(a) SetA: Desired resources (b) SetA: Allocated resources (c) SetB: Desired resources (d) SetB: Allocated resources

Fig. 5. How *NEPTUNE* manages resource contentions.

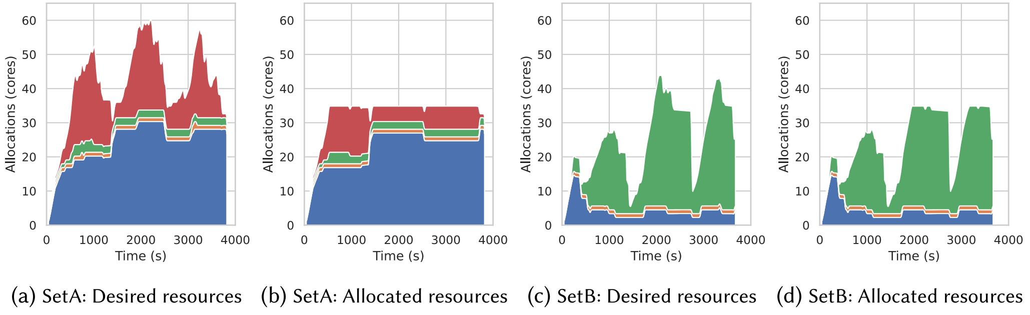
configurations. When we used $k = 10^2$ and $k = 10^3$, *NEPTUNE* was able to allocate resources to let no requests exceed the desired response time. Furthermore, the average response times were close to the target one of PI controllers (i.e., 10s) and the standard deviations were low. This highlighted the precise and robust control of *NEPTUNE*.

5.3 RQ2: Autoscaling and Resource Contention

The second set of experiments aimed to assess the *Contention Manager* proposed at Node level when dealing with multiple applications at the same time. We used the same infrastructure described in Section 5.2 (i.e., master and worker nodes on a *c5.2xlarge* and a *c5.9xlarge* virtual machines, respectively). We also used a sinusoidal workload configured with $MinUsers = 1$, $MaxUsers = 250$, $StartTime = 60s$, $Period = 1200s$, and $Duration = 3600s$, but instead of running one function at a time, we ran multiple functions concurrently. The workload was designed to have three peaks (at seconds 700, 1900, and 3100), and its goal was to consume all available resources. Then, we grouped the functions into two sets. *SetA* comprised four functions: *graph-bfs*, *dynamic-html*, *thumbnailer*, and *video-processing*. The workload was split as follows: 1% to *video-processing* and 33% to each other function. *SetB* comprised three functions: *graph-mst* and *compression*, *video-processing*; 1% of the workload again was assigned to *video-processing* and the rest was split equally between the other two functions (i.e., 49.5% to each function). Note that we decided to forward a significantly smaller percentage of requests to *video-processing* since it requires more resources per request.

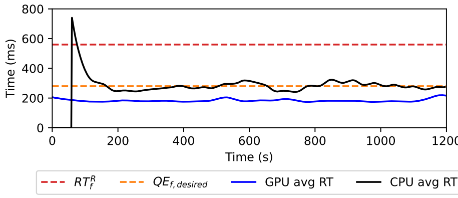
Figure 5 presents one (randomly chosen) run of the experiments. Figures 5(a) and 5(b) (5(c) and 5(d)) illustrate the desired resources (i.e., the resources that should be allocated if nodes had infinite resources) and actual allocated resources (over time) for functions in *SetA* (*SetB*). The figures show the benefits of resource contention. In correspondence to workload peaks, allocated resources are similar to desired ones. This behavior is due to the proportional policy employed by the *Contention Manager*, which assigns resources in a fair manner (i.e., *NEPTUNE* splits resources according to the demand). We can also observe that desired resources only exceeded a few times the maximum amount of available resources (i.e., 36 cores), which means that *NEPTUNE* suggests resource allocations that, in the worst case, are close to feasible.

For comparison, we ran the same experiments with HPA, an autoscaling solution that does not implement any mechanism to solve resource contentions. HPA assign resources by using a FIFO (First In First Out) policy, that is, resources are assigned to the first function that asks for them. Figure 6 shows obtained results. *SetA* functions did not have a fair resource allocation compared with their demand. During workload peaks, the amount of desired resources for function *video-processing* (Figure 6(a)), was very similar to that of function *graph-bfs*, but the amount of allocated



(a) SetA: Desired resources (b) SetA: Allocated resources (c) SetB: Desired resources (d) SetB: Allocated resources

Fig. 6. How HPA manages resource contentions.



(a) Response time over time.

Function		Response Time (ms) ↓		
		GPU	CPU	Total
resnet-a	μ	180.4	282.3	200.3
	σ	8.8	42.3	16.6
resnet-b	μ	183.3	288.9	202.2
	σ	9.5	66.1	14.3

(b) Response time.

Fig. 7. Resnet: CPU and GPU executions.

resources (Figure 6(b)) was significantly smaller. The second and third workload peaks, at seconds 1900 and 3100, respectively, highlight this behavior: less than 10 cores were allocated to *video-processing* and more than 20 to *graph-bfs*. Figures 6(c) and 6(d) exemplify a similar behaviour with *SetB* functions.

Autoscaling approaches—with no contention mechanisms—might propose unfeasible resource allocations. This severely degrades the effectiveness of these solutions. For example, after some 1900 seconds, HPA would like to have 60 cores for *SetA* functions, while only 36 cores were available.

5.4 RQ3: GPU Management

The third set of experiments focused on GPU management for computationally intensive functions. The experiments were conducted in a heterogeneous environment using two *c5.xlarge* virtual machines and one *g4dn.xlarge* virtual machine that was equipped with a GPU. The functions used for the experiments were *resnet-a* and *resnet-b*, both embed the *ResNet* neural network in inference mode and were configured to exploit the available GPU. For each run, we use a ramp workload with $MinUsers = 10$, $MaxUsers = 30$, $StartTime = 10s$, $RampTime = 30s$, and $Duration = 1200s$.

Figure 7 shows the results of a single execution of the experiments since the applications behaved similarly in all the runs. In addition, Figure 7(a) only presents the average response time of the GPU and of the CPUs with resnet-a since the two applications showed a similar trend.

The workloads executed on the GPU produced an almost constant response time and never violated the threshold. Initially, the incoming workload was small, and the routing policies redirected the traffic to the GPU. It handled the whole incoming workload till its full utilization, for some 50 seconds. At that point, the Community level quickly reacted by updating the routing policies, and part of the workload was forwarded to the CPUs. The response time shows an initial peak and a

few violations caused by the cold start of the functions. Then, the Node level came into play and allocated the CPUs to bring the response time close to the set point.

Figure 7(b) reports the average response time for both resnet-a and resnet-b. It contains the average response time when functions are running on GPUs, on CPUs, and the aggregated results. The GPU kept the response time almost constant and caused no violations. In particular, the average response times were 180ms and 183ms for resnet-a and resnet-b, three times smaller than the threshold. The use of CPUs led to a wider distribution. In fact, the standard deviations of CPU instances were 42.3ms and 66.1ms for resnet-a and resnet-b, respectively. PI controllers managed the CPU cores: they introduced an initial transient period needed to adjust the initial core allocation.

The CPU-only replicas of resnet-a and resnet-b served 98.3% and 100%, respectively, of the requests within the set response time. The GPU handled 70% of the requests and the CPUs handled the rest. The total number of violations in both cases is close to 0. The results also say that the transparent GPU management provided by *NEPTUNE* allowed the two functions (*resnet-a* and *resnet-b*) to seamlessly share the same GPU. Figure 7(b) shows that the sharing did not degrade the performance of either application since both applications presented similar distributions of response times. Moreover, the interplay between GPU and CPU instances helps keep the average response time as low as 200.3ms for resnet-a and 202.2ms for resnet-b.

5.5 RQ4: Placement and Routing

The fourth set of experiments aimed to evaluate the placement and routing solution proposed in *NEPTUNE*. Specifically, the objective was to evaluate different configurations ($\alpha \in \{0, 0.5, 1\}$) of the Community level and analyze the tradeoff between minimizing the network delay and the number of MEC nodes needed to manage the incoming workload. The experiments were conducted using a multi-region environment with four *m5.xlarge* virtual machines placed across four different areas (i.e., Area A, B, C, and D). The experiments employed the functions provided in SeBS and we used the three different workloads (i.e., W_{cs}, W_{td}, W_{tel}) generated from real-world traces and with a duration of 30 minutes. The workload takes into consideration that function *video-processing* is very resource demanding, only 1% of the users were designed to request that function, while the remaining users were randomly spread across all other functions (i.e., 19.8% to each function).

Table 5 reports the results (mean and standard deviation are reported as "$\mu \pm \sigma$") of the experiments with workload W_{cs}. Concerning network delay and $\alpha = 0$, *NEPTUNE* aimed only to place function instances close to the incoming workload. In fact, we can observe that under this configuration, *NEPTUNE* achieved the lowest network delays: from 17.1ms (function *compression*) to 199.1ms (function *graph-mst*). Increasing α made *NEPTUNE* also consider the cost factor of running the MEC infrastructure but also led to higher network delay. For example, with $\alpha = 0.5$ and $\alpha = 1$, the average times spent by requests in the network for function *graph-mst* were, respectively, 265.5ms and 341.2ms (more than the required response time $RT_f^R = 200$ms).

Higher network delays also led to higher response time violations. For instance, with $\alpha = 0$, only 1.5% of the requests to function *compression* exceeded the required response time, while with $\alpha = 0.5$ and $\alpha = 1$, the violation rates were, respectively, 4.8 and 5.9 times higher. Regarding allocated cores to functions, there were no statistically significant differences between the three configurations. This means that, on average, the three configurations of *NEPTUNE* allocated a similar overall amount of cores to functions. However, if we consider the amount of nodes used to serve the incoming workload, we can observe that with $\alpha = 1$, *NEPTUNE* activated on average only 1.3 nodes, while configurations $\alpha = 0$ and $\alpha = 0.5$ required 3 and 2.5 times more active nodes.

In comparison with competitor approaches, results show that *NEPTUNE* configured with $\alpha = 0.5$ was able to outperform VSVBP, MCF, and CR-EUA in most aspects. For instance, the average network delay of *NEPTUNE* with $\alpha = 0.5$ was 113.8ms, while VSVBP, MCF, and CR-EUA generated,

Table 5. Results with Different Settings of α when Managing Workload W_{cs} and Comparison with Competitors

	NEPTUNE			*Competitors*		
	$\alpha = 0$	$\alpha = 0.5$	$\alpha = 1$	VSVBP	MCF	CR-EUA
Function	**Network Delay (ms) ↓**					
compression	**17.1 ± 0.1**	34.2 ± 2.9	34.5 ± 3.5	57.9 ± 4.5	51.6 ± 1.6	32.7 ± 4.4
dynamic-html	**28.8 ± 1.7**	32.0 ± 0.6	51.5 ± 6.5	67.9 ± 3.5	31.5 ± 1.8	37.4 ± 1.0
graph-bfs	**34.0 ± 5.4**	38.0 ± 1.5	41.0 ± 4.6	54.3 ± 4.1	76.1 ± 5.3	39.6 ± 5.2
graph-mst	**199.1 ± 13.9**	265.5 ± 3.8	341.2 ± 10.8	323.9 ± 14.8	275.6 ± 14.6	264.0 ± 3.2
thumbnailer	**102.2 ± 9.7**	130.4 ± 3.5	177.6 ± 0.5	221.6 ± 1.3	184.7 ± 4.3	127.5 ± 6.9
video-processing	**155.3 ± 8.9**	182.7 ± 5.4	274.3 ± 9.3	255.6 ± 7.0	204.5 ± 4.7	198.9 ± 0.7
	Response Time Violation (%) ↓					
compression	**1.5 ± 0.7**	7.2 ± 0.1	8.9 ± 1.9	11.8 ± 0.3	11.7 ± 0.2	9.3 ± 1.8
dynamic-html	**4.1 ± 1.9**	5.7 ± 1.2	13.2 ± 3.1	17.8 ± 0.1	5.8 ± 0.9	12.0 ± 3.0
graph-bfs	**15.6 ± 2.6**	16.1 ± 3.7	15.7 ± 2.3	12.5 ± 1.9	22.5 ± 3.0	15.0 ± 3.5
graph-mst	**65.2 ± 0.9**	91.5 ± 6.3	98.5 ± 1.8	95.1 ± 3.3	94.7 ± 3.4	89.6 ± 0.5
thumbnailer	**33.9 ± 1.5**	62.1 ± 6.0	54.4 ± 6.6	81.5 ± 5.5	65.9 ± 7.2	69.3 ± 2.6
video-processing	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
	Core Allocation (millicores) ↓					
compression	340.8 ± 30.0	**327.4 ± 25.4**	362.9 ± 29.8	559.4 ± 49.5	623.8 ± 9.8	566.5 ± 13.2
dynamic-html	511.0 ± 17.4	**513.3 ± 14.8**	522.7 ± 30.3	681.0 ± 39.3	445.8 ± 15.0	677.6 ± 24.1
graph-bfs	**639.0 ± 16.9**	654.2 ± 10.1	668.0 ± 17.7	853.0 ± 53.3	872.9 ± 34.8	842.6 ± 6.0
graph-mst	1073.4 ± 18.8	1078.3 ± 48.6	**978.2 ± 23.0**	1307.0 ± 19.2	874.7 ± 12.2	1364.0 ± 64.4
thumbnailer	**608.0 ± 7.0**	616.8 ± 34.0	650.6 ± 32.4	818.4 ± 19.8	872.1 ± 8.8	757.0 ± 49.0
video-processing	823.8 ± 21.1	820.7 ± 2.1	**813.3 ± 23.0**	1938.6 ± 49.7	1940.2 ± 36.1	1943.9 ± 77.0
	Active Nodes (nodes) ↓					
overall	3.9 ± 0.1	3.2 ± 0.4	**1.3 ± 0.1**	1.6 ± 0.2	1.4 ± 0.2	2.7 ± 0.6

Best results are highlighted in bold.

respectively, 43.7%, 20.7%, and 2.5% higher network delays. Competitors also served fewer requests within the desired response time (19.7%, 9.9%, and 12.3% more violations) while allocating more resources (53.5%, 40.3%, and 53.3% more). However, from the cost point of view, the competitors required on average only 1.6, 1.4, and 2.7 active nodes to handle the incoming workload, while *NEPTUNE* with $\alpha = 0.5$ required 3.2 nodes.

This behavior can be explained by the fact that competitors model the problem as a user allocation problem and aim to maximize the number of users allocated to as few MEC nodes as possible. However, they do not take into consideration some aspects of real-world workloads. For instance, they do not consider the scenario where new users appear at runtime or existing users move across different areas. Furthermore, the absence of mechanisms that minimize service disruption also contributed to higher network delays and violation rates.

Table 6 reports an overall view of the results of *NEPTUNE* and competitors when we used the three different workloads (W_{cs}, W_{td} and W_{tel}). The results confirm what was observed with workload W_{cs}. For instance, if we consider the three configurations of *NEPTUNE*, the best network delays, and response time violations, are obtained with $\alpha = 0$ in almost all cases. Furthermore, obtained allocated cores are similar across the three α configurations, while only with $\alpha = 1$ the number of active nodes was reduced.

Compared with competitors, *NEPTUNE* obtained a more reliable behaviour. For example, with workload W_{td}, which resulted to be the most difficult workload to handle, *NEPTUNE* with $\alpha = 0.5$ exceeded the required response time only 30.5% of the time, while VSVBP, MCF, and CR-EUA obtained violation rates equal to 52.0%, 35.3%, and 60.2%, respectively. Overall, *NEPTUNE* obtained 44.5% lower network delays, 42.5% less response time violations while allocating 50.8% fewer resources with respect to its competitors.

Table 6. Overall Results of *NEPTUNE* and Competitors with Workloads W_{cs}, W_{td} and W_{tel}

	Network Delay (ms) ↓			Response Time Violations (%) ↓			Core Allocations (millicores) ↓			Active Nodes (nodes) ↓		
	W_{cs}	W_{td}	W_{tel}	W_{cs}	W_{td}	W_{tel}	W_{cs}	W_{td}	W_{tel}	W_{cs}	W_{td}	W_{tel}
NEPT, $\alpha = 0$	**89.4**	**99.1**	**94.1**	**20.1**	31.2	**27.5**	3996.0	**3870.0**	3981.0	3.9	3.9	3.9
NEPT, $\alpha = 0.5$	113.8	113.2	126.2	30.4	**30.5**	30.2	4010.7	3888.0	**3979.2**	3.2	3.3	3.1
NEPT, $\alpha = 1$	153.4	204.6	140.3	31.8	56.8	31.9	**3995.6**	3891.6	4027.8	**1.3**	**1.3**	**1.3**
VSVBP	163.5	281.9	153.2	36.4	52.0	46.3	6157.4	6182.5	6142.7	1.6	1.2	1.5
MCF	137.3	101.6	194.1	33.4	35.3	57.4	5629.5	6236.1	6590.0	1.4	1.9	1.4
CR-EUA	116.7	313.0	177.2	34.2	60.2	58.4	6151.6	6082.2	6135.0	2.7	2.3	2.2

Best results are highlighted in bold.

5.6 Threats to Validity

We conducted a large set of experiments and showed that *NEPTUNE* can minimize network delay, reduce response times, efficiently allocate resources, solve resource contentions, and manage CPUs and GPUs transparently. However, we must highlight some threats that may constrain the validity of obtained results [60]:

Internal Threats. The experiments were conducted on a cloud infrastructure, which allowed us to emulate significant latencies among nodes by utilizing different regions and availability zones. While this approach provided valuable insights into the performance of the proposed system, it is essential to consider the potential impact of other aspects that can exist in real-world MEC infrastructures. For example, in cloud environments network performance is generally not subject to unforeseen fluctuations. Conversely, at the edge, the network conditions could be more dynamic and less reliable. To mitigate these kind of issues, our theoretical framework and prototype have been designed to take into account most of such dynamic conditions. For example, the inter-delay among nodes is constantly measured and updated, thus decisions are taken without any assumptions on (nearly) constant network performance. Moreover, in our experiments, we used real-world workloads that capture the characteristics of edge use cases (e.g., taxis moving around different areas). Additionally, the functions we took from the literature do not provide a recommended value for the response time RT_f^R. To obtain this value, we used an iterative process. We started from $50ms$, and we incremented the value by $50ms$ until the function was able to serve at least 50% of requests in a time equal to $RT_f^R/2$.

External Threats. Some of our assumptions may limit the generalization of the experiments. Since *NEPTUNE* adopts the serverless paradigm, it assumes that functions be either stateless or depend on an external database. Currently, *NEPTUNE* only partially models database interactions. The delays introduced by reading from and writing to a database are modeled at the Node level as a non-controllable stationary disturbance of the response time (e.g., a Gaussian noise). We previously demonstrated [4] that *NEPTUNE* can efficiently handle functions that rely on a database with a precision that is similar to those without data dependencies.

Moreover, our approach has only been tested on a single cloud provider (AWS) posing a potential limitation to the generalizability of our study. However, we have designed *NEPTUNE* to be cloud-agnostic, not relying on AWS-specific features. Thus, we expect that similar performance could be obtained across different cloud providers.

Finally, the assumption that GPUs should be prioritized over CPUs may not be valid in scenarios where cost is determined by electricity or battery availability. In such instances, a careful evaluation is needed between the computational speed-up of GPUs and their associated higher costs.

However, these objectives are beyond the scope of this article and will be addressed in our future work.

Construct and Conclusion Threats. We adopted a special-purpose monitoring infrastructure to gather metrics during the experiments. Such infrastructure collects data about latency and resource allocation. However, those data can be subject to uncertainty and noise, which could compromise the measurement process. To demonstrate the validity of our claims, and reduce uncertainty and noise, we repeated all experiments five times, and obtained results are statistically robust.

6 RELATED WORK

Recently, there has been a significant increase in the need for applications with low latency requirements, such as those necessary for safety-critical functions. To meet these demands, it has become necessary to deploy these applications on an edge infrastructure to facilitate immediate response to user requests [57]. This led to a surge in research addressing the associated challenges of edge computing [15, 43]. It is now recognized that many architectural solutions that are effective in the cloud domain do not align well with the unique nature of the edge [46]. Notably, conventional solutions, such as centralized architectures, often overlook the geographical placement of computing nodes. Thus, the definition of strategies for application placement and request routing in edge systems has become a major research area [7, 21].

To the best of our knowledge, *NEPTUNE* is the first solution that provides: an easy-to-use serverless interface, optimal function placement and routing policies, in-place vertical scaling of functions, and transparent management of GPUs and CPUs. The relevant related works only focus on specific aspects of the problem. For example, Wang et al. [58] propose *LaSS*, a framework for latency-sensitive edge computations built on top of Kubernetes and Openwhisk. LaSS models the resource allocation problem by using a *M/M/c FCFS* queuing model. It provides a fair-share resource allocation algorithm, similar to the *Contention Manager* employed by *NEPTUNE*, and two reclamation policies for freeing allocated resources. LaSS is the most similar solution to *NEPTUNE*, but it lacks the minimization of network overhead (i.e., function placement) and active nodes, and does not support GPUs explicitly. Furthermore, the approach is not fully compatible with the Kubernetes API. Kubernetes is only used to deploy OpenWhisk. Functions run natively on top of the container runtime (e.g., Docker) and resources are vertically scaled by bypassing Kubernetes. This approach, also adopted in cloud computing solutions [6, 45], is known to create inconsistent state representations between the container runtime and the orchestrator [3].

Ascigil et al. [1] formulate the resource provisioning problem for serverless functions in hybrid edge-cloud systems by using *MIP*. They propose both fully-centralized (orchestration) approaches, where a single controller is in charge of allocating resources, and fully-decentralized (choreography) ones, where controllers placed across the network operate independently with no coordination. Compared with *NEPTUNE*, they focus on minimizing the number of unserved requests, and they assume to serve each request in a fixed amount of time (single time slot). However, this assumption is not easy to ensure in edge computing: nodes may be equipped with different types of hardware and produce disparate response times. *NEPTUNE* handles this aspect by transparently exploiting GPUs and CPUs.

Multiple approaches focus on placement and routing at the edge [8, 17, 41, 44, 47]. One of the most widely used techniques, also employed by *NEPTUNE*, is to model the service placement and workload routing as a *(Mixed) Integer Programming* problem [26, 27]. Ma et al. [36] formulate the problem as a ***Mixed Integer Linear Programming*** (**MILP**) problem, and they propose a solution capable of maximizing the revenue (i.e., utilization) of edge nodes. However, compared with *NEPTUNE*, their solution does not consider network delays and, as a consequence, they cannot

constrain latency by placing applications close to the users. Tong et al. [54] model a MEC network as a hierarchical tree of geo-distributed servers and formulate the problem as a two-step ***Mixed Nonlinear Integer Programming*** (**MNIP**). In particular, their approach aims to maximize the amount of served requests by using optimal service placement and resource allocation. The effectiveness of the approach is verified through formal analysis and large-scale trace-based simulations. They assume that workloads follow some known stochastic models (Poisson distribution) and that arrival rates are independent and identically distributed. This last assumption may not be realistic in the context of edge computing where workloads are often unpredictable and may significantly deviate from assumed distributions. *NEPTUNE* does not share these assumptions and uses fast control-theoretical planners to mitigate volatility and unpredictability in the short term.

To cope with dynamic workloads, Tan et al. [53] propose an online algorithm for workload dispatching and scheduling without any assumption about distribution. However, since their approach only focuses on routing requests, they cannot always keep network delays low, especially when edge clients move from one location to another. Liu et al. [34] propose a solution in the public safety domain that first serves the user requests with higher criticality (i.e., level of danger). This solution comprises two algorithms: one for small-scale edge topologies and one for big-scale ones. Despite not being explicitly designed to minimize network delays, the approach can effectively manage this aspect as long as user criticalities take networking latencies and desired response times into consideration.

Nomadic workloads are explicitly addressed, for example, by Leyva-Pupo et al. [28]. They present a solution based on an ***Integer Linear Programming*** (**ILP**) problem with two objective functions: one for nomadic users and one for static ones. Furthermore, since the problem is known to be NP-hard, they use heuristic methods to compute a sub-optimal solution. Sun et al. [52] propose a service migration solution based on *MIP* to keep the computation as close to the user as possible. In particular, they consider different factors that contribute to migration costs (e.g., required time and resources). However, the two previous solutions exploit virtual machines, known for their large image sizes and long start-up times, that make service migration a costly operation. *NEPTUNE*, as other approaches in the literature [37, 58, 64], uses containers that are lighter and faster to scale.

Only a few solutions have been proposed for GPU management in the context of edge computing. For example, Subedi et al. [51] mainly focus on enabling GPU-accelerated edge computations without considering latency-critical aspects such as placing applications close to users.

7 CONCLUSIONS AND FUTURE WORK

The article presents *NEPTUNE*, a serverless-based solution for managing latency-sensitive applications deployed on large-scale edge topologies. *NEPTUNE* minimizes network overhead by adopting a dedicated function placement mechanism and optimized routing policies, copes with fluctuating workloads by employing a dynamic resource allocation mechanism, and supports transparent management of both CPUs and GPUs. *NEPTUNE* has been implemented in a prototype built on top of K3S, a popular container orchestrator for the edge. We conducted an extensive evaluation to demonstrate the feasibility of the approach and assess *NEPTUNE* against state-of-the-art solutions.

NEPTUNE can be extended in different ways. In particular, our future work comprises improvements on the placement strategy and on the resource allocation mechanism to exploit function dependencies [32] and to embed state migration capabilities. To improve the performance of *NEPTUNE*, we also plan to adopt workload predictors to anticipate future demand [24] and Bayesian optimization approaches [14, 50] to find optimal response times automatically.

REFERENCES

[1] Onur Ascigil, Argyrios G. Tasiopoulos, Truong Khoa Phan, Vasilis Sourlas, Ioannis Psaras, and George Pavlou. 2022. Resource provisioning and allocation in function-as-a-service edge-clouds. *IEEE Transactions on Service Computing* 15, 4 (2022), 2410–2424.

[2] David Balla, Csaba Simon, and Markosz Maliosz. 2020. Adaptive scaling of kubernetes pods. In *Proceedings of the Network Operations and Management Symposium*. IEEE, 1–5.

[3] Luciano Baresi, Davide Yi Xian Hu, Giovanni Quattrocchi, and Luca Terracciano. 2021. KOSMOS: Vertical and horizontal resource autoscaling for kubernetes. In *Proceedings of the International Conference on Service-Oriented Computing*, Vol. 13121. Springer, 821–829.

[4] Luciano Baresi, Davide Yi Xian Hu, Giovanni Quattrocchi, and Luca Terracciano. 2022. NEPTUNE: Network- and GPU-aware management of serverless functions at the edge. In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 144–155.

[5] Luciano Baresi, Danilo Filgueira Mendonça, and Giovanni Quattrocchi. 2019. PAPS: A framework for decentralized self-management at the edge. In *Proceedings of the International Conference on Service-Oriented Computing*, Vol. 11895. Springer, 508–522.

[6] Luciano Baresi and Giovanni Quattrocchi. 2020. COCOS: A scalable architecture for containerized heterogeneous systems. In *Proceedings of the International Conference on Software Architecture*. IEEE, 103–113.

[7] Julian Bellendorf and Zoltán Ádám Mann. 2020. Classification of optimization problems in fog computing. *Elsevier Future Generation Computer Systems* 107, C (2020), 158–176.

[8] David Bermbach, Jonathan Bader, Jonathan Hasenburg, Tobias Pfandzelter, and Lauritz Thamsen. 2021. AuctionWhisk: Using an auction-inspired approach for function placement in serverless fog platforms. *Wiley Software Practice and Experience* 52, 5 (2021), 1–49.

[9] Victor Campmany, Sergio Silva, Antonio Espinosa, Juan Carlos Moure, David Vázquez, and Antonio M. López. 2016. GPU-based pedestrian detection for autonomous driving. In *Proceedings of the International Conference on Computing Science*, Vol. 80. Elsevier, 2377–2381.

[10] Junguk Cho, Karthikeyan Sundaresan, Rajesh Mahindra, Jacobus E. van der Merwe, and Sampath Rangarajan. 2016. ACACIA: Context-aware edge computing for continuous interactive applications over mobile networks. In *Proceedings of the International on Conference on emerging Networking EXperiments and Technologies*. ACM, 375–389.

[11] Thomas Heide Clausen and Philippe Jacquet. 2003. Optimized link state routing protocol (OLSR). *RFC* 3626 (2003), 1–75.

[12] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the International Middleware Conference*. ACM, 64–78.

[13] Xavier Dutreilh, Nicolas Rivierre, Aurélien Moreau, Jacques Malenfant, and Isis Truck. 2010. From data center resource allocation to control theory and back. In *Proceedings of the International Conference on Cloud Computing*. IEEE, 410–417.

[14] Nicolò Felicioni, Andrea Donati, Luca Conterio, Luca Bartoccioni, Davide Yi Xian Hu, Cesare Bernardis, and Maurizio Ferrari Dacrema. 2020. Multi-objective blended ensemble for highly imbalanced sequence aware tweet engagement prediction. In *Proceedings of the Rec. Sys. Challenge 2020*. ACM, 29–33.

[15] Ana Juan Ferrer, Joan Manuel Marquès, and Josep Jorba. 2019. Towards the decentralised cloud: Survey on approaches and challenges for mobile, ad hoc, and edge computing. *ACM Computing Survey* 51, 6 (2019), 111:1–111:36.

[16] Domenico Grimaldi, Valerio Persico, Antonio Pescapè, Alessandro Salvi, and Stefania Santini. 2015. A feedback-control approach for resource management in public clouds. In *Proceedings of the Global Communications Conference*. IEEE, 1–7.

[17] Songtao Guo, Bin Xiao, Yuanyuan Yang, and Yang Yang. 2016. Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing. In *Proceedings of the International Conference on Computer Communications*. IEEE, 1–9.

[18] Yan Guo, Shangguang Wang, Ao Zhou, Jinliang Xu, Jie Yuan, and Ching-Hsien Hsu. 2020. User allocation-aware edge cloud placement in mobile edge computing. *Wiley Software Practice and Experience* 50, 5 (2020), 489–502.

[19] Akhil Gupta and Rakesh Kumar Jha. 2015. A survey of 5G network: Architecture and emerging technologies. *IEEE Access* 3 (2015), 1206–1232.

[20] Jiwei Huang, Ming Wang, Yuan Wu, Ying Chen, and Xuemin Shen. 2022. Distributed offloading in overlapping areas of mobile-edge computing for internet of things. *IEEE Internet of Things Journal* 9, 15 (2022), 13837–13847.

[21] Congfeng Jiang, Xiaolan Cheng, Honghao Gao, Xin Zhou, and Jian Wan. 2019. Toward computation offloading in edge computing: A survey. *IEEE Access* 7 (2019), 131543–131558.

[22] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A.

Patterson. 2019. Cloud programming simplified: A berkeley view on serverless computing. https://arxiv.org/abs/1902.03383

[23] Patrick Kalmbach, Andreas Blenk, Wolfgang Kellerer, Rastin Pries, Michael Jarschel, and Marco Hoffmann. 2019. GPU accelerated planning and placement of edge clouds. In *Proceedings of the International Conference on Networked Systems*. IEEE, 1–3.

[24] Jitendra Kumar and Ashutosh Kumar Singh. 2018. Workload prediction in cloud using artificial neural network and adaptive differential evolution. *Elsevier Future Generation Computer Systems* 81, C (2018), 41–52.

[25] Indika Kumara et al. 2021. SODALITE@RT: Orchestrating applications on cloud-edge infrastructures. *Journal of Grid Computing* 19, 3 (2021), 29.

[26] Phu Lai, Qiang He, Mohamed Abdelrazek, Feifei Chen, John G. Hosking, John C. Grundy, and Yun Yang. 2018. Optimal edge user allocation in edge computing with variable sized vector bin packing. In *Proceedings of the International Conference on Service-Oriented Computing*, Vol. 11236. Springer, 230–245.

[27] Phu Lai, Qiang He, John Grundy, Feifei Chen, Mohamed Abdelrazek, John G. Hosking, and Yun Yang. 2022. Cost-effective app user allocation in an edge computing environment. *IEEE Transactions on Cloud Computing* 10, 3 (2022), 1701–1713.

[28] Irian Leyva-Pupo, Alejandro Santoyo-González, and Cristina Cervelló-Pastor. 2019. A framework for the joint placement of edge service infrastructure and user plane functions for 5G. *Sensors* 19, 18 (2019), 3975.

[29] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. 2010. CloudCmp: Comparing public cloud providers. In *Proceedings of the Internet Measurement Conference*. ACM, 1–14.

[30] Ping-Min Lin and Alex Glikson. 2019. Mitigating cold starts in serverless platforms: A pool-based approach. https://arxiv.org/abs/1903.12221

[31] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md. Enamul Haque, Lingjia Tang, and Jason Mars. 2018. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 751–766.

[32] Wei-Tsung Lin, Chandra Krintz, and Rich Wolski. 2018. Tracing function dependencies across clouds. In *Proceedings of the International Conference on Cloud Computing*. IEEE, 253–260.

[33] Ensheng Liu, Liping Zheng, Qiang He, Phu Lai, Benzhu Xu, and Gaofeng Zhang. 2023. Role-based user allocation driven by criticality in edge computing. *IEEE Transactions on Service Computing (Just Accepted)* 15, 5 (2023), 1–14.

[34] Ensheng Liu, Liping Zheng, Qiang He, Benzhu Xu, and Gaofeng Zhang. 2023. Criticality-awareness edge user allocation for public safety. *IEEE Transactions on Service Computing* 16, 1 (2023), 221–234.

[35] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. 2019. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE* 107, 8 (2019), 1697–1716.

[36] Zhi Ma, Sheng Zhang, Zhiqi Chen, Tao Han, Zhuzhong Qian, Mingjun Xiao, Ning Chen, Jie Wu, and Sanglu Lu. 2022. Towards revenue-driven multi-user online task offloading in edge computing. *IEEE Transactions on Parallel and Distributed Systems* 33, 5 (2022), 1185–1198.

[37] Omogbai Oleghe. 2021. Container placement and migration in edge computing: Concept and scheduling models. *IEEE Access* 9 (2021), 68028–68043.

[38] Quoc-Viet Pham, Fang Fang, Vu Nguyen Ha, Md. Jalil Piran, Mai Le, Long Bao Le, Won-Joo Hwang, and Zhiguo Ding. 2020. A survey of multi-access edge computing in 5G and beyond: Fundamentals, technology integration, and state-of-the-art. *IEEE Access* 8 (2020), 116974–117017.

[39] Michal Piorkowski, Natasa Sarafijanovic-Djukic, and Matthias Grossglauser. 2009. A parsimonious model of mobile partitioned networks with clustering. In *Proceedings of the International Communication Systems and Networks and Workshops*. IEEE, 1–10.

[40] Jon Postel. 1981. Internet control message protocol. *RFC* 777 (1981), 1–14.

[41] Konstantinos Poularakis, Jaime Llorca, Antonia Maria Tulino, Ian J. Taylor, and Leandros Tassiulas. 2019. Joint service placement and request routing in multi-cell mobile edge computing networks. In *Proceedings of the Conference on Computer Communications*. IEEE, 10–18.

[42] Peter-Christian Quint and Nane Kratzke. 2018. Towards a lightweight multi-cloud DSL for elastic and transferable cloud-native applications. In *Proceedings of the International Conference on Cloud Computing and Service Science*. SciTePress, 400–408.

[43] Philipp Raith, Stefan Nastic, and Schahram Dustdar. 2023. Serverless edge computing - where we are and what lies ahead. *IEEE Internet Computing* 27, 3 (2023), 50–64.

[44] Philipp Raith, Thomas Rausch, Schahram Dustdar, Fabiana Rossi, Valeria Cardellini, and Rajiv Ranjan. 2022. Mobility-aware serverless function adaptations across the edge-cloud continuum. In *Proceedings of the International Conference on Utility and Cloud Computing*. IEEE, 123–132.

[45] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. 2019. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. In *Proceedings of the International Conference on Cloud Computing*. IEEE, 33–40.

[46] Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. 2023. Serverless functions in the cloud-edge continuum: Challenges and opportunities. In *Proceedings of the International Conference on Parallel, Distributed and Network-Based Processing*. IEEE, 321–328.

[47] Gabriele Russo Russo, Tiziana Mannucci, Valeria Cardellini, and Francesco Lo Presti. 2023. Serverledge: Decentralized function-as-a-service for the edge-cloud continuum. In *Proceedings of the International Conference on Pervasive Computing and Communications*. IEEE, 131–140.

[48] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: Workload autoscaling at Google. In *Proceedings of the EuroSys Conference*. ACM, 16:1–16:16.

[49] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking functions to warm the serverless cold start. In *Proceedings of the International Middleware Conference*. ACM, 1–13.

[50] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the Advances in Neural Information Proceedings Systems*. 2960–2968.

[51] Piyush Subedi, Jianwei Hao, In Kee Kim, and Lakshmish Ramaswamy. 2021. AI multi-tenancy on edge: Concurrent deep learning model executions and dynamic model placements on edge devices. In *Proceedings of the International Conference on Cloud Computing*. IEEE, 31–42.

[52] Xiang Sun and Nirwan Ansari. 2016. PRIMAL: PRofIt maximization avatar placement for mobile edge computing. In *Proceedings of the International Conference on Communications*. IEEE, 1–6.

[53] Haisheng Tan, Zhenhua Han, Xiang-Yang Li, and Francis C. M. Lau. 2017. Online job dispatching and scheduling in edge-clouds. In *Proceedings of the International Conference on Computer Communications*. IEEE, 1–9.

[54] Liang Tong, Yong Li, and Wei Gao. 2016. A hierarchical edge cloud architecture for mobile computing. In *Proceedings of the International Conference on Computer Communications*. IEEE, 1–9.

[55] Athanasios Tsipis and Konstantinos Oikonomou. 2022. Joint optimization of social interactivity and server provisioning for interactive games in edge computing. *Computing Networks* 212, C (2022), 109028.

[56] Abhishek Verma, Hussam Qassim, and David Feinzimer. 2017. Residual squeeze CNDS deep learning CNN model for very large scale places image recognition. In *Proceedings of the Annual Ubiquitous Computing, Electronics and Mobile Communication Conference, 2017*. IEEE, 463–469.

[57] Michael Vierhauser, Rebekka Wohlrab, and Stefan Rass. 2022. Towards cost-benefit-aware adaptive monitoring for cyber-physical systems. In *Proceedings of the Conference on Communications and Network Security*. IEEE, 1–6.

[58] Bin Wang, Ahmed Ali-Eldin, and Prashant J. Shenoy. 2021. LaSS: Running latency sensitive serverless computations at the edge. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*. IEEE, 239–251.

[59] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. 2018. Peeking behind the curtains of serverless platforms. In *Proceedings of the USENIX Annual Technical Conference*. USENIX, 133–146.

[60] Claes Wohlin, Martin Höst, and Kennet Henningsson. 2006. Empirical research methods in web and software engineering. In *Proceedings of the Web Engineering*. 409–430.

[61] Jierui Xie, Boleslaw K. Szymanski, and Xiaoming Liu. 2011. SLPA: Uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process. In *Proceedings of the International Conference on Data Mining Workshops*. IEEE, 344–349.

[62] Jing Yuan, Yu Zheng, Xing Xie, and Guangzhong Sun. 2011. Driving with knowledge from the physical world. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*. ACM, 316–324.

[63] Xu Zhang, Hao Chen, Yangchao Zhao, Zhan Ma, Yiling Xu, Haojun Huang, Hao Yin, and Dapeng Oliver Wu. 2019. Improving cloud gaming experience through mobile edge computing. *IEEE Wireless Communications* 26, 4 (2019), 178–183.

[64] Ao Zhou, Shangguang Wang, Shaohua Wan, and Lianyong Qi. 2020. LMM: Latency-aware micro-service mashup in mobile edge computing environment. *Neural Computing Applications* 32, 19 (2020), 15411–15425.

[65] Qian Zhu and Gagan Agrawal. 2012. Resource provisioning with budget constraints for adaptive applications in cloud environments. *IEEE Transactions on Service Computing* 5, 4 (2012), 497–511.

Received 5 November 2022; revised 31 October 2023; accepted 1 November 2023