

Dependency-aware Resource Allocation for Serverless Functions at the Edge

Luciano Baresi, Giovanni Quattrocchi, and Inacio Ticongolo

Dipartimento di Elettronica, Informazione e Bioingegneria,
Politecnico di Milano, Italy
{name.surname}@polimi.it

Abstract. Serverless computing allows developers to break their code into small components, known as functions, which are automatically managed by a service provider. Being lightweight and modular, serverless functions have been increasingly employed in edge computing, where quick responses and adaptability are key to meeting strict latency requirements. In particular, edge nodes are intrinsically resource-constrained, and efficient resource allocation strategies are crucial for optimizing their usage. Different approaches exist in the literature, but they often overlook the dependencies among functions, that is, how and when functions invoke other functions, obtaining suboptimal results.

This paper presents NEPTUNE⁺, a dependency-aware resource (CPU cores) allocation solution for serverless functions deployed at the edge. The approach extends NEPTUNE, an existing framework for managing edge infrastructures, with a new theoretical model and control algorithm that take dependencies into account function. We evaluated NEPTUNE⁺ by using three applications and it is able to allocate up to 42% fewer cores compared to NEPTUNE.

Keywords: serverless · edge computing · function dependencies · resource allocation

1 Introduction

Complex applications are increasingly built as sets of independent components, such as microservices [4] to foster agility and speed in both development and runtime management. This high degree of modularization allows for independent management, but complicates communication among components and affects their performance. For this reason, understanding the logical dependencies among components is essential for the efficient management of the system [10].

Serverless computing is arising as a new instance of such highly modularized architectural paradigms [16]. It promotes the creation of applications as a collection of “small” functions [17], which are usually executed in lightweight containers to ensure their fast and efficient management [6]. Serverless functions are designed to be independently developed, deployed, and scaled automatically by a service provider. This high degree of flexibility allows for fast re-configuration

and scaling in response to changes in the system workload and contributes to overall system agility.

Given these characteristics, serverless functions have been increasingly employed in edge computing [16]. In this context, applications are often constrained by strict latency requirements. The inherent agility and ability to rapidly scale individual components allow serverless platforms to quickly adapt to workload fluctuations, facilitate the prompt execution of functions, and allow the system to meet latency requirements more effectively [21]. However, it is essential to acknowledge that serverless functions can depend on other functions, which can significantly impact their performance and management [9].

In the last few years, serverless computing has been widely studied as means to improve the management of applications deployed on edge infrastructures [12]. Some approaches tackle the intelligent placement of edge functions on resource-limited nodes [6,8]; others focus on optimizing resource allocation [19]. Yet, these approaches often overlook function dependencies, a crucial factor for performance modeling [22].

This paper introduces NEPTUNE⁺, a solution that focuses on resource allocation for serverless functions deployed at the edge. NEPTUNE⁺ extends an existing edge framework, called NEPTUNE [2], which allows for the smart placement and allocation of serverless functions but does not consider function dependencies. NEPTUNE⁺ extends the theoretical model of NEPTUNE by proposing i) a new formalization of the problem that encodes function dependencies as an annotated Direct Acyclic Graph (DAG), and ii) a novel control algorithm that exploits the function dependency graph to save resources. A comprehensive empirical evaluation compared NEPTUNE and NEPTUNE⁺ by means of three benchmark applications. Obtained results show that NEPTUNE⁺ allocates up to 42% fewer cores than NEPTUNE, with comparable performance in terms of response times.

The rest of the paper is organized as follows. Section 2 introduces NEPTUNE and highlights its limitations. Section 3 describes our solution, along with the new problem formulation and control algorithm. Section 4 presents the evaluation and discusses the results. Section 5 surveys the related work, and Section 6 concludes the paper.

2 NEPTUNE in a nutshell

Serverless computing is the driver of a significant paradigm shift that frees developers from infrastructure management [17], and some approaches [22,7] explored this paradigm for deploying and managing applications in edge infrastructures. To the best of our knowledge, only NEPTUNE [2] provides a comprehensive and holistic management approach that considers network partitioning, placement, request routing, and the combined dynamic allocation of memory, CPUs, and GPUs.

NEPTUNE requires the code of the function to deploy, a threshold (service level agreement or SLA) on its response time, and the identification of the

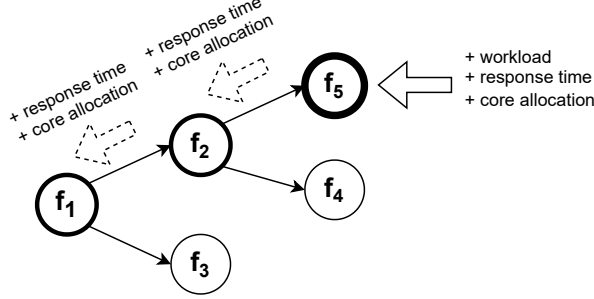


Fig. 1: Example application with function dependencies.

memory required for proper execution. The management exploits a three-level hierarchy: topology, community, and node. The global network *topology* is split into a set of independent *communities*. Each community is composed of edge *nodes* (or servers) that are close to each other, that is, their network inter-delays are smaller than a set threshold. Each community is managed by a dedicated controller that takes into account user mobility, workload provenance, and the memory requirements for each function. This controller exploits an optimization problem based on Mixed-Integer Programming to calculate the best placement of function instances and a set of routing policies that minimize network latency. Function placement implies deciding how many function instances are needed for each used function and the best node to host each of them. Since each node cannot always host all the instances and handle all the incoming workload, NEPTUNE uses routing policies to compute the request fraction to be routed to other nodes. The same formulation is used to first handle the workload that can be accelerated through GPUs, and then, the remaining workload is assigned to CPUs exclusively.

Whereas GPUs and memory are entirely managed by the community controller, the node level controller oversees the proper execution of requests on CPUs, making sure that each function instance is provisioned with enough cores to complete executions within the given SLA. A lightweight Proportional Integer (PI) controller is attached to each function instance f_i with the goal of keeping the response time close to a given set point:

$$sp_i = \alpha * SLA_i \quad (1)$$

where α is a scaling parameter ($0 < \alpha \leq 1$). The more α is close to 1, the more the response time is kept close to SLA, with a risk for potential violations. Conversely, if α is significantly lower than 1, the controller ensures better performance, but more resources are needed. Therefore, α represents a tunable trade-off between performance and resource utilization.

2.1 Limitations

NEPTUNE does not consider function dependencies, which can lead to inefficient resource allocation. Let us consider, for example, the application in Figure 1. This application consists of five functions, f_1, f_2, f_3, f_4, f_5 , with their dependencies outlined in a directed acyclic graph (DAG): f_1 depends on f_2 and f_3 , while f_2 depends on f_4 and f_5 .

If we suppose that function f_5 is supposed to manage a workload spike, its response time suddenly increases, and its local node level controller is prompted to augment allocated cores. While f_5 's controller stabilizes the situation and brings the response time closer to the set point, the response times of f_2 , and then of f_1 , also grow, given their dependency on f_5 .

The inefficiency of NEPTUNE lies in the behavior of the controllers associated with f_2 and f_1 's: higher response times, due to the slow responses from f_5 , imply increasing the cores allocated to f_1 and f_2 . This reaction is redundant, as the issue does not stem from either f_2 or f_1 , but the bottleneck is f_5 . Instead of allocating extra cores to f_2 and f_1 , these resources would have been better utilized to speed up f_5 .

This is why NEPTUNE must be improved/extended to address this limitation and fix redundant allocations: CPU cores must be allocated efficiently even in the presence of dependent serverless functions.

3 Solution

NEPTUNE⁺ extends NEPTUNE with a new theoretical model and control algorithm to efficiently allocate resources to serverless functions with dependencies. In particular, NEPTUNE⁺ aims to improve the allocation of CPUs cores at the node level in light of the limitations described in Section 2.1, while it inherits from NEPTUNE its placement strategy that minimizes network delays and the management of GPUs and memory.

3.1 Theoretical Model

Let F be a set of serverless functions whose dependencies are encoded as a DAG where nodes are the function instances and edges are the invocations between functions. This DAG is assumed to be either manually defined by the user or automatically generated utilizing network or log analysis techniques [13].

We do not consider invocation cycles (i.e., we employed DAGs for modeling dependencies), in line with the recommendations against their use, as suggested by Fontana et al. [11]. A cycle denotes a situation where a function indirectly relies on its own output to commence its execution, forming an untenable loop. This could lead to an endless cycle of executions or deadlock scenarios that are unmanageable cost-wise on real serverless platforms [20].

NEPTUNE⁺ considers the response time rt_i of a function $f_i \in F$ as follows:

$$rt_i = lrt_i + ert_i \quad (2)$$

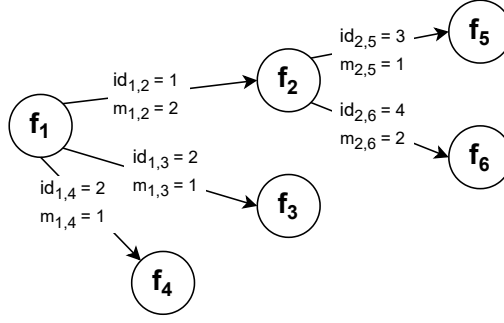


Fig. 2: Example of annotated DAG.

where lrt_i is the *local* response time spent for executing its code, that is, the set of instructions that implements the function without considering external calls to other functions, and ert_i is the *external* response time, that is, the time spent for invoking other functions. The external response time depends on the type of dependency between f_i and another function f_j . First, the invocation could be either *sequential* or *parallel*. In the former case, the invocation is synchronous, that is, other invocations must wait for its completion. The latter allows it to be executed in parallel with some other invocations. More formally, each edge in the DAG is annotated with an identifier $id_{i,j}$. If two edges $E_{i,j}$ and $E_{i,k}$, which represent the invocations of f_j and f_k , respectively, in f_i , have different identifiers, the invocations are executed sequentially. If they are annotated with the same value, they are called in parallel.¹ Moreover, each edge is also annotated with a *multiplier* $m_{i,j}$, which denotes how many times such invocation is executed within the same function call [22]. Figure 2 shows an example of an annotated DAG with 5 functions. Function f_1 sequentially calls function f_2 (i.e., $id_{1,2}$ is unique) two times during its execution (i.e., $m_{1,2} = 2$) —for example, at the beginning and at the end of the computation. Moreover, f_1 invokes f_3 and f_4 in parallel (i.e., $id_{1,3} = id_{1,4} = 2$) each only once (i.e., $m_{1,3} = m_{1,4} = 1$). Finally, f_2 invokes sequentially f_5 (once) and f_6 (two times) since the edge identifiers are unique. The external response time is defined as follows:

$$ert_i = \sum_{j \in S} m_{i,j} * rt_j + \sum_{P \in \bar{P}} \max(m_{i,j} * rt_j, \forall j \in P) \quad (3)$$

where S is the set of functions that f_i invokes sequentially, \bar{P} is the set of the subsets of functions that f_i invokes in parallel, and P represents each subset (i.e., a group of parallel invocations). In essence, the external response time is equal to the sum of all sequential invocations plus the sum of all the longest

¹ Note that our approach allows a function f_i to invoke another function f_j both in sequential and parallel mode by having multiple, properly annotated, edges between i and j (as in *multigraphs* [1]). Herein, we did not detail such edge cases to keep our formalization as simple as possible.

response times of each parallel group taking into account how many times each dependency is called (i.e., $m_{i,j}$).

In NEPTUNE⁺, function instances can receive requests from users (i.e., direct invocations) and/or from other functions (i.e., external invocations). More formally, the total amount of requests r_i^w received by f_i in a set time window w is defined as follows:

$$r_i^w = r_{users \rightarrow f_i}^w + \sum_{j \in E_{*,i}} m_{j,i} * r_j^w \quad (4)$$

where $r_{users \rightarrow f_i}^w$ is the total amount of direct invocations of f_i in w and $E_{*,i}$ is the set of edges from a source node (i.e., a function f_j) to f_i .

3.2 Control Algorithm

While NEPTUNE's node controllers consider each function independently of the others, NEPTUNE⁺ adapts the control strategy by considering the dependency DAG. The control algorithm we employ distinguishes between *entrypoint* functions and *externally invoked* functions. The former can be invoked by users, that is, $r_{users \rightarrow f_i}^w > 0$ for some w , whereas the latter are *only* called by other functions in the DAG².

NEPTUNE⁺ allows users to define an SLA for each entrypoint function f_i . If f_i is only called by users and not by other functions, such an input is mandatory, whereas it is optional if f_i is also invoked by other functions (as a consequence of a direct invocation of another entrypoint).

NEPTUNE⁺ inherits from NEPTUNE its PI controllers, which compute resource allocations without synchronizing with one another. The main difference between the two approaches is that NEPTUNE monitors and controls the response rt_i of each function f_i without discriminating between local and external response times, whereas NEPTUNE⁺ focuses on the local response time lrt_i . The intuition behind this design is that rt_i is affected by the response times of other functions (external invocations), which results in the problems described in Section 2.1. In contrast, lrt_i depends solely on the resource allocation of f_i and allows for a more fine-grained and optimized control strategy. Since NEPTUNE⁺ exploits lrt_i , it cannot simply reuse Equation 1 to define *local set points* for the PI controllers, since SLA_i is defined as an upper bound of the *total* response rt_i . Thus, NEPTUNE⁺ computes, at design time, the local set points lsp_i of each function f_i by considering: i) the user-defined SLAs for entrypoint functions, ii) the dependency among functions, and iii) the *weight* of each function within the DAG. Intuitively, higher weights correspond to higher set points since such functions are considered more complex compared to others.

In particular, the weight of a function f_i is calculated by using the *nominal response time* nrt_i and the *nominal local response time* $nlrt_i$: nrt_i and $nlrt_i$ are

² Note that, in this context, the invocation is *external* with respect to the execution environment of the invoked function (as for the use of “external” in *external response time*).

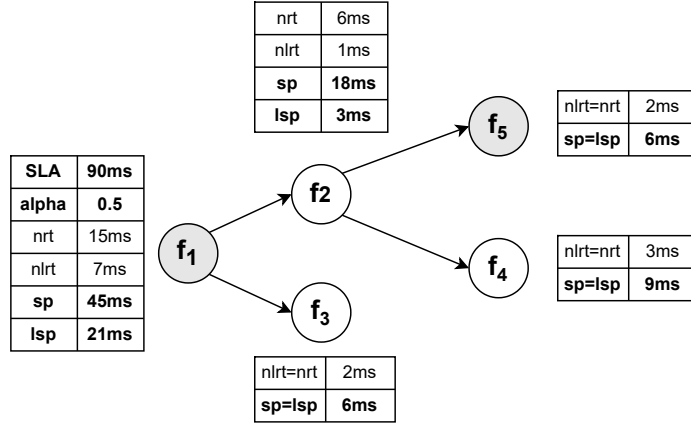


Fig. 3: Example of local set point computation.

modeled, respectively, by using the formulas used to calculate rt_i (Equation 2) and lrt_i (Equation 3). However, whereas rt_i and lrt_i are measured at runtime under user-generated workloads, nrt_i and $nlrt_i$ are measured during a profiling phase while considering the system in a quiescent state (i.e., without saturation or request queue). Each function is profiled with a static core allocation (e.g., 1 core) and by sending one request at a time, waiting for the previous to finish. To avoid considering cold starts, the measurement starts after a warm-up period. These two metrics are used to understand the complexity of functions and their dependencies in a “controlled” state and are used to properly compute set points.

To explain how we calculate local set points, we employ the same application described in Section 2 and Figure 3 shows the main calculations. Functions f_1 and f_5 are depicted in light gray and are the two entrypoints of the application. For the sake of simplicity, we consider all the dependencies as sequential and with all multipliers $m_{i,j}$ equal to 1. Thus, we avoided reporting the DAG annotation.

Let f_i be an entrypoint function with a user-defined SLA_i , and a set point sp_i defined as in Equation 1. In the example, only f_1 has a user-defined SLA that is equal to $90ms$. Since α is set to 0.5, the set point sp_1 is equal to $45ms$. The local set point lsp_i is defined as:

$$lsp_i = sp_i * \frac{nlrt_i}{nrt_i} \quad (5)$$

Thus, it follows that, in the example, the local set point of function f_1 is equal to $21ms$. Moreover, for each function f_j that is invoked by f_i the set point sp_j is then defined as:

$$sp_j = \frac{sp_i}{m_{i,j}} * \frac{nrt_j}{nrt_i} \quad (6)$$

Algorithm 1 Core allocation

```

1:  $lrt_i \leftarrow getLocalResponseTime(f_i)$ 
2:  $err \leftarrow lsp_i^{-1} - lrt_i^{-1}$ 
3:  $P \leftarrow gain_P * err$ 
4:  $I \leftarrow I + gain_I * err$ 
5:  $cores \leftarrow P + I$ 
6:  $cores \leftarrow \min(cores_{MAX}, \max(cores_{MIN}, cores))$ 
7:  $allocateCores(f_i, cores)$ 

```

The setpoint sp_i is used to compute the local set point of f_i and the set points sp_j of all the dependencies. Given that sp_j is intended to consider a single invocation, the calculation is divided by $m_{i,j}$.

In the example, the dependencies of f_1 are f_2 and f_3 . This means that, ideally, the sum of the local response time of f_1 and the (total) response times of f_2 and f_3 should be equal to set point sp_1 . Since f_3 has no dependencies, its nominal response time nrt_3 is equal to its nominal local response time $nlrt_3$. Intuitively, since nrt_3 is around 3 times lower than $nlrt_1$ the set point of f_3 (6ms) is roughly 3 times lower than the local set point of f_1 (21ms). Instead, f_2 depends on f_4 and f_5 , and its set point sp_2 is set to 18ms with a nominal response time nrt_2 equal to 6ms. This means, in turn, that the sum of the local response time of f_2 and the response times of f_4 and f_5 should be kept equal to 18ms.

Recursively, the local set points of each dependency are calculated by using Equation 5. For example, the set point sp_2 is used to calculate the local set point of f_2 (3ms) along with the set points of f_4 (6ms) and f_5 (9ms). Note that the case of a function without dependencies is the base case of the recursive procedure, and its set point is equal to its local set point, such as for f_3 , f_4 , and f_5 . Finally, to further optimize the resource allocation, the set points of parallel dependencies (i.e., edges with the same source node and identifiers) are calculated as in Equation 6. However, after the calculation, the set point of each dependency is set to be equal to the maximum of the parallel group. This means that even though siblings could complete execution faster, they are slowed down with higher set points to match the slowest function of the parallel group. This strategy does not affect the overall response time of the application and allows for saving resources.

Proportional-Integral Control. After the previous steps, each function is provided with a local set point lsp_i . As in NEPTUNE, each function f_i is equipped with a PI controller. In NEPTUNE⁺, the controller for a function f_i monitors only the local response time lrt_i and allocates cores to meet lsp_i . Without any synchronization and thanks to the computations above, if all the controllers are able to meet their local set points, user-defined SLAs are fulfilled. The control algorithm we used, adapted from NEPTUNE, is reported in Algorithm 1.

The procedure is invoked at every control period for each function instance f_i . The local response time (obtained at line 1) and the local set point are used to compute the error err . The higher the error is, the higher the mismatch

between the local set point and the actual measured local response time (line 2). The proportional contribution (P) is equal to the proportional gain $gain_P$ multiplied by err (line 3). The integral contribution (I) is the sum of the previous actions and the error times the integral gain $gain_I$ (line 4). Both $gain_P$ and $gain_I$ are tuning parameters of the controller and can be set using different well-known heuristics [5]. The core allocation is computed as the sum of P and I (line 5) properly scaled according to the minimum ($cores_{MIN}$) and maximum ($cores_{MAX}$) allowed amount of cores (line 6). Finally, the allocation is enacted at line 7.

4 Evaluation

Our evaluation was aimed to compare NEPTUNE⁺ against NEPTUNE in the absence of bottlenecks and when bottlenecks occur.

We ran all the experiments on a MacBook Pro equipped with 4 cores and 16GB of RAM running macOS Ventura (version 13.2.1). To test the two systems, we relied on an existing simulator called RAS (Resource Allocation Simulator), created by the same authors of NEPTUNE [3]. The RAS simulator was originally used to evaluate the performance of the control algorithms of NEPTUNE against industrial approaches. We extended the simulator ³ in two ways: i) we adapted the code to support function dependencies, and ii) we implemented the novel theoretical model and control algorithm.

The experiments used three different applications. The first two are benchmarks widely used in the literature [15], namely *hotel reservation*⁴ and *sockshop*⁵). The first is a serverless application that mimics a hotel reservation website, whereas the second is an online e-commerce application that exploits a microservice architecture that was converted to serverless functions by the authors of NEPTUNE [2]. *Hotel reservation* includes four functions (2 entrypoints), and it is characterized by a DAG with an average out-degree of 3 edges, an average in-degree of 1 edge, and all the dependencies have type *sequential*. *Sockshop* includes 7 functions (5 entrypoints) with an average *out - degree* of 6 edges, an average *in - degree* of 1 edge, and one third of the dependencies have type *sequential* while the remaining two thirds have type *parallel*. We also created a more complex scenario, application *complex*, by synthesizing a DAG of 25 functions (6 entrypoints), with an average out-degree of 2 edges, an average in-degree of 1 edge, and roughly balanced sequential and parallel dependencies.

We repeated each experiment 10 times. In each test, we simulated executions of 20 minutes each, and we collected, for each function, the average (μ) and the standard deviation (σ) of three metrics: response times (RT) in milliseconds, core allocations (C) in millicores, and percentage of SLA violations (V).

The tests employed workloads similar to the ones used to evaluate NEPTUNE in [3,2]. In particular, each entrypoint function was stimulated with either a *ramp*

³ Source code available at <https://doi.org/10.5281/zenodo.8174489>

⁴ <https://github.com/vhive-serverless/vSwarm/tree/main/benchmarks/hotel-app>

⁵ <https://github.com/microservices-demo/microservices-demo>

or a step *workload*. We employed ramps that start from 10 requests and added one request every second up to 100 (as in [2]) and randomly generated steps that vary the workload every 50 seconds in a range between 20 and 120 requests. We also simulated bottlenecks by changing the random step to a number of requests that ranges between 800 and 6000.

For NEPTUNE we set an SLA for each function, whereas for NEPTUNE⁺ we only set them for entrypoints, since our approach is able to automatically calculate the set points for all the other functions. For *sock-shop*, we employed the same SLAs reported in the original NEPTUNE paper [2]. For *hotel reservation* and *complex application*, we set the SLAs to double their nominal response times. The nominal response times of *hotel reservation* were obtained by profiling each function, while for *complex application* we generated them randomly.

We configured both NEPTUNE and NEPTUNE⁺ the same way. We employed a value for α equal to 0.5 for each function with SLA as in [2]. We derived the values of $gain_P$ and $gain_I$ through manual tuning (again, as in [2]).

To be sure that the simulator was aligned with realistic results and that our modifications did not affect its accuracy, we executed a preliminary experiment. We simulated the same tests on NEPTUNE run in [2] with application *sockshop* (i.e., same workload and configuration). We collected the results and compared them with those reported in the paper. We observed that on average the differences were minimal: 0.3% for response times and 4.3% for core allocation.

4.1 Performance without bottlenecks

Table 1 shows the results obtained by NEPTUNE (N) and NEPTUNE⁺ (N^+). For the first two applications, the table lists the tested functions along with their SLAs. For application *complex*, it only shows averages due to lack of space. Functions marked with a * are entrypoints. Row *overall* reports the averages over all functions.

If we focus on the part *without bottlenecks*, one can observe that NEPTUNE⁺ consistently outperforms NEPTUNE in many cases.

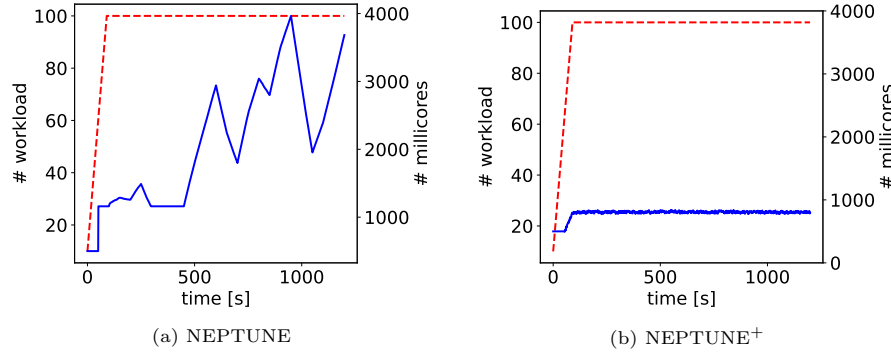
For example, if we consider function *order* of *sockshop* application, NEPTUNE⁺ yields a significantly more efficient resource allocation compared to NEPTUNE (788 millicores allocated by NEPTUNE⁺ against 1133 millicores allocated by NEPTUNE). The response time of NEPTUNE⁺ (291.3 ms) is closer to the 300 ms set point ($\alpha * SLA$ with $\alpha = 0.5$ and $SLA = 600ms$) compared to the result obtained by NEPTUNE (211.2 ms). This means that NEPTUNE⁺ does not need to over-provision CPU cores to meet the user-defined SLA and only allocates needed resources. Overall, with benchmark *sockshop*, NEPTUNE⁺ demonstrates a more efficient performance by reducing required millicores from 510 to 388, marking a 24% reduction, while only having a small increase in average response time (85.6 vs. 63.1 ms) and no SLA violations. Note that, faster response times can be also obtained by NEPTUNE⁺ by simply lowering the set points.

Table 1: Results without and with bottlenecks.

<i>f</i>	SLA		without bottlenecks						with bottlenecks					
			RT		V		C		RT		V		C	
			<i>N</i>	<i>N+</i>	<i>N</i>	<i>N+</i>	<i>N</i>	<i>N+</i>	<i>N</i>	<i>N+</i>	<i>N</i>	<i>N+</i>	<i>N</i>	<i>N+</i>
<i>hotel reservation</i>														
search*	118	μ	56.6	62.6	0	0	327	282	65	78.2	0	0	2719	282
		σ	0	0.1	0	0	0.2	0	0	0	0	0	4	0
profile*	36	μ	17.3	17.2	0	0	343	346	33	33	46.6	46.5	2285	2290
		σ	0	0	0	0	0.2	0.2	0	0	0.1	0.1	0.7	0.7
geo	27	μ	12.9	12.8	0	0	243	245	12.9	12.8	0	0	243	245
		σ	0	0	0	0	0.1	0.1	0	0	0	0	0.1	0.1
rate	34	μ	16.3	16.3	0	0	295	295	16.3	16.3	0	0	295	295
		σ	0	0	0	0	0.2	0.2	0	0	0	0	0.1	0.1
overall		μ	25.8	27.2	0	0	302	292	31.8	35	11.7	11.6	1386	778
<i>sockshop</i>														
orders*	600	μ	211.2	291.3	0	0	1133	788	317.3	384.7	0	0	2093	788.3
		σ	0.1	0	0	0	0	0.6	0.1	0.1	0	0	5.2	0.5
catalogue*	200	μ	50.6	72.4	0	0	126	88	50.6	72.4	0	0	126	88
		σ	0	0	0	0	0	0.1	0.1	0	0	0	0	0.1
shipping	50	μ	15.4	20.6	0	0	414	312	15.4	20.6	0	0	414	312
		σ	0	0	0	0	0	0.1	0	0	0	0	0	0.2
users*	50	μ	24.1	29.7	0	0	154	127	24.1	29.7	0	0	154	127
		σ	0	0	0	0	0.1	0.1	0	0	0	0	0.1	0.1
payment*	50	μ	13.7	14.4	0	0	605	578	13.7	14.4	0	0	605	578
		σ	0	0	0	0	0	0.6	0	0	0	0	0	0.5
cart-utils	200	μ	58.8	79	0	0	511	372	58.9	79	0	0	511	372
		σ	0	0	0	0	0	0.4	0	0	0	0	0	0.3
cart-del*	200	μ	67.8	92	0	0	628	450	185.8	185.4	45.9	45.9	1527	1530
		σ	0	0	0	0	0	0.3	0.1	0.1	0	0	0.2	0.2
overall		μ	63.1	85.6	0	0	510	388	95.1	112.3	6.6	6.6	776	542
<i>complex</i>														
overall		μ	240.0	263.7	0	0	4627	3013	313.2	357.5	11.5	15.3	6530	3760

The trend is similar with more complex applications (i.e., *complex*): NEPTUNE⁺ yields a more efficient allocation (3013 vs. 4627 millicores), with a 27% improvement, no SLA violations, and comparable to NEPTUNE response times.

Conversely, the two approaches provide similar performance with benchmark *hotel reservation* except for function *search* where NEPTUNE⁺ is slightly more efficient in terms of core allocation. This can be attributed to the application’s simple DAG and its limited amount of dependencies. This result demonstrates that NEPTUNE⁺ does not introduce any performance degradation in scenarios where dependencies are not a critical factor.

Fig. 4: Results for function *order* (*sockshop*).

4.2 Performance with bottlenecks

Table 1 also shows the results obtained when managing bottlenecks (created as explained above). As for application *hotel reservation*, we raised the number of requests for function *profile*, leading to a significant amount (around 47%) of SLA violations obtained by both NEPTUNE and NEPTUNE⁺. Such a bottleneck inevitably raises the response time of *search*, which directly depends on *profile*. Since NEPTUNE⁺ only considers local response times, our solution is able to properly manage this function by only allocating 282 millicores on average, while NEPTUNE raises the average core allocation to 2719 millicores, that is, some 90% higher than NEPTUNE⁺. The response times are comparable: 65ms with NEPTUNE and 78.2ms NEPTUNE⁺. The other functions, not affected by the bottleneck, showed performance similar to the ones observed in the experiments described in the previous section. Overall with this application, NEPTUNE⁺ obtained a slightly higher average response time (35ms vs. 31.8ms) and a 44% lower average core allocation (778 vs. 1386 millicores).

We observed similar results also with *sockshop*. In this case, we created a bottleneck in function *cart-del* as demonstrated by the high number of violations obtained by the two approaches. The results reported for function *order*, which depends on *cart-del*, clearly show the benefits of NEPTUNE⁺. While NEPTUNE⁺ results in a higher response time (384.7 vs. 317.3), it also significantly reduces the number of cores used, allocating only 788.3 millicores against 2093 (62% improvement). This is clearly shown in Figure 4, where NEPTUNE’s controller for function *order* is unstable due to the bottleneck in *cart-del* and reaches a peak of some 4000 allocated millicores. In contrast, NEPTUNE⁺ keeps its allocation roughly stable at around 800 millicores after the initial ramp. Overall, as for *sockshop*, NEPTUNE⁺ obtained a core allocation that is almost 30% better (lower) on average than NEPTUNE with comparable response times (equal SLA violations).

Application *complex* suggests that the more complex an application becomes, the more efficient NEPTUNE⁺ is: 3760 vs. 6530 millicores, with a 42% improvement at the cost of only 4.8% more SLA violations.

By taking into account function dependencies, NEPTUNE⁺ efficiently allocates resources across diverse benchmarks and scenarios. Conversely, NEPTUNE, without dependency awareness, tends to over-provision resources. This results in faster, yet less optimized, response times that only rarely lead to fewer SLA violations. NEPTUNE’s behavior is partly due to its inability to maintain set points, resulting in an over-speeding that is not beneficial in most of the cases. Instead, NEPTUNE⁺ provides more precise control and offers a more convenient trade-off between resource efficiency and response times. If faster response times are required, NEPTUNE⁺ users can simply define stricter SLAs or lower the value of α to obtain a more responsive system.

5 Related Work

The problem of managing microservices or serverless functions deployed on edge infrastructures has been already studied in the literature [21,2]. Such approaches tackle component placement, routing, and resource management, but only a few of them take function dependency into account [14,18,22].

He et al. [14] introduce a novel approach for deploying microservices to edge servers by taking into account their intricate dependencies with the goal of optimizing response time. They do not consider resource allocation but only component placement. Therefore, the amount of CPU cores to obtain a certain response time is not optimized. In contrast, NEPTUNE⁺ considers the trade-off between allocated resources and response time. They also do not consider parallel and multiple invocations of the same components.

Xu et al. [22] propose a solution for optimizing the placement at the edge of serverless functions with dependencies. They also take into account stateful computation and network delays. Similarly, Ashraf et al. [18] propose SONIC, a solution that aims to optimize the performance and operation cost of serverless applications by deciding the best function placement for exchanging data. Applications are abstracted as DAGs as in NEPTUNE⁺. These approaches select the best path for exchanging data by considering data size, function dependencies, and network state. They are complementary to NEPTUNE⁺ since they exploit dependencies for optimizing data exchange and do not consider resource allocation.

Moving to runtime resource management [4,7,21,23], these studies either disregard function dependencies entirely, as in the case of [21], or they utilize a probabilistic approach to pre-allocate functions, such as [4,7,23]. For instance, Daw et al. [7] introduce Xanadu, which uses a directed acyclic graph of dependencies and a probabilistic model to identify the most likely execution paths. To reduce the overhead from the cascading cold-start of functions, it pre-allocates resources (i.e., containers) for the most probable path in response to each function call. Similarly, Kraken [4] estimates the request volume for each function

and uses these estimates to deploy the necessary containers. By batching multiple requests, Kraken utilizes fewer resources than Xanadu: under moderate to heavy load, Xanadu deploys nearly 32% more containers than Kraken [4]. However, both Xanadu and Kraken’s probabilistic approaches can lead to resource over- or under-provisioning if the estimated probability distribution does not reflect the actual workload fluctuations over time. Furthermore, neither solution considers resource allocation for functions with dependencies in the event of a bottleneck as NEPTUNE⁺ does.

Conversely, Wang et al. [21] present LaSS, a platform for managing the latency of serverless computations. LaSS uses a dynamic resource allocation strategy based on workload variations and a queuing model. A weighted fair-share resource allocation strategy is employed to prevent overload and maintain the desired response time. While this work makes a significant contribution by mitigating SLA violations and over-allocation of resources, the authors do not consider function dependencies, which could lead to inefficient allocations. Once more, NEPTUNE⁺ uses control theory to only allocate the necessary resources to functions, based on the number of requests and defined SLA, and allows for a more efficient resource usage.

6 Conclusions

This paper presents NEPTUNE⁺, a dependency-aware resource allocation solution for serverless functions deployed on edge infrastructures. We extended NEPTUNE by developing a new theoretical model and control algorithm that exploit dependencies to efficiently allocate CPU cores to serverless functions. The evaluation shows that NEPTUNE⁺ outperforms the original framework up to 42% in terms of resource allocation. In the future, we will improve our solution by also considering the placement of dependency-aware functions.

References

1. VK Balakrishnan. *Graph theory*, volume 1. McGraw-Hill New York, 1997.
2. Luciano Baresi, Davide Yi Xian Hu, Giovanni Quattrocchi, and Luca Terracciano. Neptune: Network-and gpu-aware management of serverless functions at the edge. In *Proc. of the Sym. on Software Engineering for Adaptive and Self-Managing Sys. (SEAMS)*, pages 144–155, 2022.
3. Luciano Baresi and Giovanni Quattrocchi. A simulation-based comparison between industrial autoscaling solutions and cocos for cloud applications. In *Int. Conf. on Web Services*, pages 94–101, 2020.
4. Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proc. of the ACM Sym. on Cloud Comp., SoCC ’21*, page 153–167. ACM, 2021.
5. Rakesh P Borase, DK Maghade, SY Sondkar, and SN Pawar. A review of pid control, tuning methods and applications. *Int. J. of Dynamics and Control*, 9:818–827, 2021.

6. Gustavo André Setti Cassel, Vinicius Facco Rodrigues, Rodrigo da Rosa Righi, Marta Rosecler Bez, Andressa Cruz Nepomuceno, and Cristiano André da Costa. Serverless computing for internet of things: A systematic literature review. *Future Generation Computer Sys.*, 128:299–316, 2022.
7. Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proc. of the 21st Int. Middleware Conf.*, Middleware ’20, page 356–370. ACM, 2020.
8. Nabil El Ioini, David Hästbacka, Claus Pahl, and Davide Taibi. Platforms for serverless at the edge: a review. In *Advances in Service-Oriented and Cloud Comp.: Int. Workshops of ESOC 2020*, pages 29–40. Springer, 2021.
9. Tarek Elgamal, Atul Sandur, Klara Nahrstedt, and Gul Agha. Costless: Optimizing cost of serverless comp. through function fusion and placement. In *2018 IEEE/ACM Sym. on Edge Comp. (SEC)*, pages 300–312, 2018.
10. Silvia Esparrachiari, Tanya Reilly, and Ashleigh Rentz. Tracking and controlling microservice dependencies: Dependency management is a crucial part of system and software design. *Queue*, 16(4):44–65, 2018.
11. Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, and Marco Zanoni. Automatic detection of instability architectural smells. In *2016 IEEE Int. Conf. on Software Maintenance and Evolution (ICSME)*, pages 433–437, 2016.
12. Phani Kishore Gadepalli, Gregor Peach, Ludmila Cherkasova, Rob Aitken, and Gabriel Parmer. Challenges and opportunities for efficient serverless comp. at the edge. In *2019 38th Sym. on Reliable Distributed Sys. (SRDS)*, pages 261–2615, 2019.
13. S. E. Ghirotti, T. Reilly, and A. Rentz. Tracking and controlling microservice dependencies. *Communications of the ACM*, 61(11):98–104, 2018.
14. Xiang He, Zhiying Tu, Markus Wagner, Xiaofei Xu, and Zhongjie Wang. Online deployment algorithms for microservice systems with complex dependencies. *IEEE Trans. on Cloud Comp.*, 2022.
15. Md Rajib Hossen, Mohammad A. Islam, and Kishwar Ahmed. Practical efficient microservice autoscaling with QoS assurance. In *Proc. of the 31st Int. Sym. on High-Performance Parallel and Distributed Comp.* ACM, jun 2022.
16. Vojdan Kjorveziroski, Sonja Filiposka, and Vladimir Trajkovik. Iot serverless computing at the edge: A systematic mapping review. *Computers*, 10(10):130, 2021.
17. Xue Li, Peng Kang, Jordan Molone, Wei Wang, and Palden Lama. Kneescale: Efficient resource scaling for serverless comp. at the edge. In *2022 22nd IEEE Int. Sym. on Cluster, Cloud and Internet Comp. (CCGrid)*, pages 180–189, 2022.
18. Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. Sonic: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conf. (USENIX ATC 21)*. USENIX Association, Virtual. forthcoming, 2021.
19. Duarte Pinto, João Pedro Dias, and Hugo Sereno Ferreira. Dynamic allocation of serverless functions in iot environments. In *2018 IEEE 16th Int. Conf. on Embedded and Ubiquitous Comp. (EUC)*, pages 1–8, 2018.
20. Davide Taibi and Valentina Lenarduzzi. On the definition of microservice bad smells. *IEEE Software*, 35(3):56–62, 2018.
21. Bin Wang, Ahmed Ali-Eldin, and Prashant Shenoy. LaSS: Running Latency Sensitive Serverless Computations at the Edge. In *Proc. of the 30th Int. Sym. on High-Performance Parallel and Distributed Comp.* ACM, jun 2021.
22. Zichuan Xu, Lizhen Zhou, Weifa Liang, Qiufen Xia, Wenzheng Xu, Wenhao Ren, Haozhe Ren, and Pan Zhou. Stateful serverless application placement in mec with function and state dependencies. *IEEE Trans. on Computers*, pages 1–14, 2023.

23. Pawel Zuk and Krzysztof Rządca. Reducing response latency of composite functions-as-a-service through scheduling. *J. of Parallel and Distributed Comp.*, 167:18–30, 2022.