A Cost-Effective Approach for End-to-End QoS Management in NFV-enabled IoT Platforms

Clovis Anicet Ouedraogo, Samir Medjiah, Christophe Chassot, Khalil Drira, Jose Aguilar

A Cost-Effective Approach for End-to-End QoS Management in NFV-enabled IoT Platforms

Clovis Anicet Ouedraogo[1], Samir Medjiah[2], Christophe Chassot[3], Khalil Drira[1], and Jose Aguilar[4]

[1] *Université Toulouse, CNRS-LAAS, Toulouse, France*
[2] *Université Toulouse, UPS, Toulouse, France*
[3] *Université Toulouse, INSA, Toulouse, France*
[4] *Universidad de Los Andes, CEMISID, Mérida, Venezuela*
{*ouedraogo, medjiah, chassot, drira*}*@laas.fr, aguilar@ula.ve*

Abstract

The systematic scaling approach is used in the literature as the only option to meet QoS requirements in response to the traffic load increase in IoT platforms. Such a solution reduces the load on the congested nodes and leads to an increase in the provisioning costs that, when measured at the scale of billions of devices, may hinder acceptability. Our objective is to enhance the existing approaches using and extending the emerging concept of Virtualized Network Functions (VNF), promoting End-to-End IoT traffic control. We start from the observation that, within Cloud-enabled platforms, the allocated resources are not fully used, making it possible to deploy traffic control network functions (NFs). Instead of systematically scaling a congested node, we dynamically deploy, on under-loaded nodes, additional NFs that exploit the available computing resources to differentiate the traffic processing level and to apply a QoS-oriented policy. The considered NFs extend the notion of VNF defined within Network Function Virtualization (NFV), to take advantage of component-based software design. To enable a possible use of the implemented functions, we formulate a multi-objective optimization problem for the efficient planning of adequate NFs and scaling actions, according to the considered multi-constrained context. The planner, called QoS4NIP (QoS for NFV-enabled IoT Platforms), is based on a Genetic Algorithm (GA). The GA metaheuristic relies on biologically inspired operators such as mutation, crossover, and selection. QoS4NIP uses the GA to generate solutions to the identified multi-objective optimization problem by making a series of improvements in an iterative process. The experiments conducted on a realistic case study show that QoS4NIP outperforms autoscaling-based approaches (65% financial cost saved).

1 Introduction

IoT ecosystem is evolving from dedicated IoT platforms[1], sketched for the requirements of a given IoT application domain, to integrated shared platforms such as oneM2M [1]. These shared platforms simultaneously support multiple application domains, such as smart grids, connected vehicles, home automation, public safety, and e-health. The next generation of the IoT ecosystem is expected to connect billions of devices with extreme heterogeneity in terms of resources (e.g., CPU and RAM) capacities and limitations, as well as software and hardware technologies for connectivity, processing, and storage.

Virtualization is a crucial technique toward the successful design and implementation of IoT platforms that handle heterogeneity. The virtualization technologies pushed by Cloud Computing are now being extended to cover the networking domain. The Network Function Virtualization (NFV) approach [2] has been introduced to solve the challenges induced by conventional middlebox technologies, such as massive and costly deployments and complex management requirements, overloads and failures, and limited upgradability. The NFV technology tackles these challenges through the virtualization of Network Functions (NFs) on Cloud-enabled infrastructures [3]. NFV allows the instantiation, configuration, and duplication of Virtualized Network Functions (VNFs) in various locations according to the NIP operator needs, which avoids installing new physical equipment [4]. In the NFV environments, autoscaling[2] is used to guarantee a certain level of QoS[3].

Motivation. Despite the recent efforts made by the industry and academia communities for QoS management in NFV-enabled IoT platforms (NIPs), we drew the following observations and conclusions. First, the

[1] An IoT platform, also known as IoT software platform or IoT middleware, implements an IoT architecture providing a variety of service to an IoT application, such as device connectivity, device management, data transfer, data management, data analytics, security, and visualization.

[2] Autoscaling is a reconfiguration scheme where the number of resources varies automatically based on the load on the platform.

[3] The term QoS refers to the measurement of the overall performance of the NIPs service. We consider the following aspects of the NIPs service: Unavailability, Throughput, and Latency.

majority of current IoT platform providers, such as Google, AWS IoT Core, or Microsoft Azure IoT, only provide "autoscaling tools" to manage resources provisioned by tenants [5]. The autoscaling scheme is automatically triggered when resource usage reaches a given threshold (e.g., CPU usage $> 80\%$). This observation drove the IoT community toward considering the cost minimization of autoscaling as an important research challenge. The second observation is that in those platforms, nodes implement the First-Come-First-Served (FCFS) [1] as a default traffic control policy. In FCFS, the traffic data coming from different IoT applications are queued together and served in the order of their arrival. For a given level of resources within a NIP's node, the FCFS processing of IoT applications' traffic crossing this node may lead to the following problem: the resource usage induced by a "greedy" IoT application can lead to QoS degradation for other IoT applications. Therefore, this would trigger a new scaling action to increase the provisioned resources. These first two observations lead to a state that the cost of the provisioned resources is not optimal. To reach an optimal solution, we propose to associate the autoscaling scheme with Traffic Control Functions (TCFs) that take into account the different QoS levels required by the IoT applications. The third observation is that the data centers have held significant Capital Expenditure but have low resource usage. For instance[4], the average CPU and memory usage rates in Google's production clusters were only 20% and 40%, respectively, in 2012 [6]. Nearly at the same time, the average CPU utilization rate of Amazon AWS EC2 was only 7% to 17% [7]. This observation leads to conclude that there are available resources that can be used to host the Traffic Control Functions (TCFs). The last observation is that in the Cloud-to-Thing continuum (C2TC) [8], the availability and capacity of the resources, namely computation, storage, and connectivity, decrease when moving from the Cloud toward Things. Typically, the IoT End Gateways, located close to Things, are small devices with limited processing, storage, and connectivity capabilities. Thus, motivated by the above conclusions and by the promises brought by the existing studies [9], one option for the deployment of TCFs within NIPs relies on the use of technologies such as NFV. However, deploying those TCFs only as VMs or OS-level containers (as required by NFV) does not cover the resources and capacities heterogeneity of future networks. To fit the decreasing resources' capacities when moving close to the Things, and to make the End-to-End deployment of NFs, a lighter solution is required. Such a solution is one of the contributions of this paper, that we name "*Application Network Function (ANF).*"

Considering these conclusions, our objective is: (1) to meet the QoS requirements of IoT applications executed on NIPs, and (2) to optimize the costs induced by a classic autoscaling scheme. For this purpose, the global approach explored in this paper is to take advantage of the different ways of deploying TCFs (i.e., via VNFs or ANFs), while taking into account nodes' resources heterogeneity. In other words, we seek to dynamically deploy (i.e., when the need arises) (1) the appropriate TCF (e.g., Dropper, Scheduler), (2) in the appropriate packaging (ANF or VNF), and (3) on the appropriate nodes of the platform (e.g., a Scheduler before a congested node, not after).

Contributions. In this context, the significant contributions of this paper are summarized below.

- We introduce the ANF concept, which relies on a tight level of isolation technique more dealing with software execution. The ANFs make possible the deployment of NFs on IoT End Gateways and support reaching the best possible use of available heterogeneous resources capacity of the C2TC. We design a collection of TCFs that we implement as VNFs and ANFs, with the aim to sustain the QoS level required by the IoT applications. We also provide the performance measurement results to get the quantitative characteristics associated with the different implementation packages (VNFs and ANFs) of the considered TCFs. We study the effects of the traffic arrival rates on the processing time and the resource usage (CPU and RAM) required for executing the TCFs. The performance measurement results are used to solve the multiobjective optimization problem introduced hereafter.

- We formulate a multiobjective optimization problem for an efficient planning scheme of TCFs deployment on the available nodes in NIPs. The designed scheme, named QoS4NIP[5], considers both TCFs deployment and scaling actions while optimizing the overall End-to-End QoS.

- We evaluate the benefits in terms of cost-saving of the solutions provided by the QoS4NIP scheme. These benefits are compared to the solutions provided by FCFS (the lazy scheme), the autoscaling scheme, and the two variants of QoS4NIP that do not consider scaling action but only TCFs (the first considers only TCFs deployed as VNFs, and second considers TCFs deployed as VNFs and ANFs). We consider a realistic case study dealing with Connected Vehicles for validation of our approach. The validation results show that our scheme, QoS4NIP, while sustaining the End-to-End QoS in NIPs, achieves the best cost-saving amongst the existing competing approaches.

Organization. The remainder of the paper is structured as follows. Section 2 discusses the related work. Section 3 develops the proposed approach and explains the key concepts. Section 4 presents the implemented

[4]No recent information is available today.

[5]The Python source of the proposed planning scheme algorithm, is available for download at https://github.com/couedrao/QoS4NIP.

TCFs and the performance evaluations of the implemented VNF and ANF concepts. Section 5 is devoted to the QoS4NIP scheme description. Section 6 demonstrates the proposed approach effectiveness for the Connected Vehicles case study. The proposed work limitations are discussed in section 7. Finally, Section 8 concludes the paper.

2 Related Work

Several fields, such as information-centric networking (ICN), overlay network, and network slicing, consider the use of NFV for the management of QoS. In this paper, since we only aim to contribute to this domain for the IoT context, we consider the reference contributions made in the literature. The following sections present a literature review analysis on *overhead minimization* for cost saving in NFV and *runtime optimization* in NFV that are essential aspects of the proposed approach.

2.1 Overhead Minimization in NFV

As presented in Section 4, in this paper, we propose the ANF concept to fully enable the deployment of NFs over the C2TC. The existing literature involves research proposals aiming to remove the massive footprint of today's NFV platforms [10–14]. In [10], the authors present the Glasgow Network Functions (GNF), a container-based NFV platform that runs and orchestrates lightweight container-based VNFs, saving core network utilization and providing lower latency. Palkar et al. [11] propose a framework (E2) for NFV packet processing. E2 provides a single coherent system for managing NFs while relieving developers from developing per-NF solutions for placement, scaling, fault-tolerance, and other functionalities. Riggio et al. [12] propose a MEC OS that supports lightweight virtualization. Yasukata et al. in [13] propose HyperNF, a high-performance NFV framework aiming at maximizing server performance when concurrently running large numbers of NFs. HyperNF implements Hypercall-based virtual I/O, placing packet forwarding logic inside the hypervisor to significantly reduce I/O synchronization overheads. Gallo et al. [14] propose a scalable NFV-based solution as a novel approach that satisfies the stated requirements for user-centric support of IoT devices. The main differences between all these frameworks and our proposal are related to the isolation of NFs. Since isolation is not a mandatory requirement in our context (the ANFs being used are considered trusted because they are only supplied by the NIP operator), ANFs have a more reduced overhead than hypervisor-based NFs.

Furthermore, the virtualization technologies proposed these studies still have significant memory and CPU requirements [15] for the C2TC. These solutions are not adapted to the common IoT End Gateways capacities. At the same time, their needs for a particular hypervisor prevent them from operating on these gateways.

2.2 Runtime Optimization for Cost-saving in NFV

Most of the work for cost saving in NFV consider the *initial planning step* or the VNFs initial development step (i.e., design-time optimization), as described in detail in [4]. However, the few works that deal with the *Runtime Optimization* for cost saving in NFV problematic, radically, consider to automatically scale the resources provisioned to the platforms without human intervention under a dynamic workload, to minimize resource cost while satisfying Quality of Service (QoS) requirement [5]. Only considering the autoscaling scheme in these studies without differentiation in the QoS levels leads to a non-optimal scheme and induces high relative costs. Ren et al. propose in [16] an adaptive autoscaling algorithm (ASA) using an analytical model to balance the cost-performance trade-off well while maintaining an acceptable level of performance for 5G mobile networks. ASA adds (or removes) VNF instances according to the number of user requests waiting. Rahman et al. propose in [17], a proactive Machine Learning (ML)-based approach to perform autoscaling of VNFs in response to dynamic traffic changes. The authors propose an ML-based planner that learns VNF (VMs and Docker containers) scaling decisions and seasonal behavior of network traffic load to generate scaling decisions ahead of time. However, the conducted experiments show that such a proposal has a high cost. Similarly, [18] investigates a reinforcement learning approach for autoscaling on NFV. Exploring a different approach, [19] proposes a negotiation-game-based autoscaling method where tenants and service providers both engage in the autoscaling decision, based on their willingness to participate, different QoS requirements, and financial gain (e.g., cost savings). Also, [19] proposes a proactive ML-based prediction method to perform Service Function Chains (SFC) autoscaling in dynamic traffic scenarios. Searching beyond the autoscaling scheme, Draxler et al. [20] propose JASPER, a fully automated approach for jointly optimizing scaling, placement, and routing for multiple network services, consisting of numerous VNFs. JASPER manages various network services that share the same substrate network, dynamically adds or removes services, and handles workload changes. On a similar line, [21] and [22] study how to optimize SFC deployment and readjustment in a dynamic situation. Authors in [22] try to jointly maximize the implementation of new users' SFCs and the adaptation of in-service users' SFCs while considering the trade-off between resource usages and operational overhead. Quang et al. in [23]

extend the SFC deployment and readjustment in a dynamic situation approach. [23] examines VNF migration by providing a model that solves the adaptive and dynamic VNF allocation problem under QoS constraints. Yu et al. [24] extended the SFC readjustment in a proactive situation approach. [24] considers load balance, energy cost, and resource usages to formulate a multi-objective optimal resource planning problem. Contributions in [16–19,23,24] do not consider network IoT End Gateways resource constraints, and this limits the applicability for NIPs, while contributions in [20–22] explicitly take it into consideration in their approaches. In [25], Cheng et al. investigate the issues of network utility degradation when implementing NFV in dynamic networks and design a proactive NFV solution from a fully stochastic perspective. Unlike existing deterministic NFV solutions that assume given network capacities and static service quality demands, their work explicitly integrates the knowledge of substantial network variations. Targeting End-to-End reliability of mission-critical traffic, Petrov et al. in [26] introduce a softwarized 5G architecture. [26] also proposes a mathematical framework to model the process of critical session transfers in a 5G access network and to quantify their impact (QoS interferences) on other user traffic flows. They implemented, in [26], a hardware prototype to investigate the practical effects of supporting mission-critical data in a 5G NFV-enabled core network.

In summary, the existing literature lacks the attention to NIPs in two perspectives. On the one hand, several [16–19, 23, 24] studies failed to take the available heterogeneous resources capacity of the C2TC into account. On the other hand, none of the current studies consider the traffic control perspective for cost saving in NIPs. The approach we propose here addresses the shortcomings of the related work mentioned below.

3 Key Concepts and Approach Overview

The main originality of our contribution consists of the combination of several changes in the autoscaling approach, with the aim to optimize the cost of QoS management for NIPs. The first change (Section 3.1) consists in considering the on-the-fly deployment of the TCFs to sustain QoS in NIPs. The second change (Section **??**) consists in considering the ANFs, in addition to the VNFs, for the TCFs deployment. The last change (Section 3.3) deals with the elaboration and implementation of a new planning scheme, QoS4NIP, which jointly optimizes scaling actions and TCFs deployment.

The frequently used abbreviations are listed in Table 1.

Table 1: Abbreviation

Abbreviation	Description
ANF	Application Network Function
C2TC	Cloud-to-Thing Continuum [8]
FCFS	First-Come-First-Served
GA	Genetic Algorithm
IoT	Internet of Things
LSL	Local Service Level
NFV-I	Network Functions Virtualization Infrastructure
NIP	NFV-enabled IoT Platform
QoS	Quality of Service
QoS4NIP	QoS for NFV-enabled IoT Platforms
TCF	Traffic Control Function
VM	Virtual Machines
VNF	Virtualized Network Function

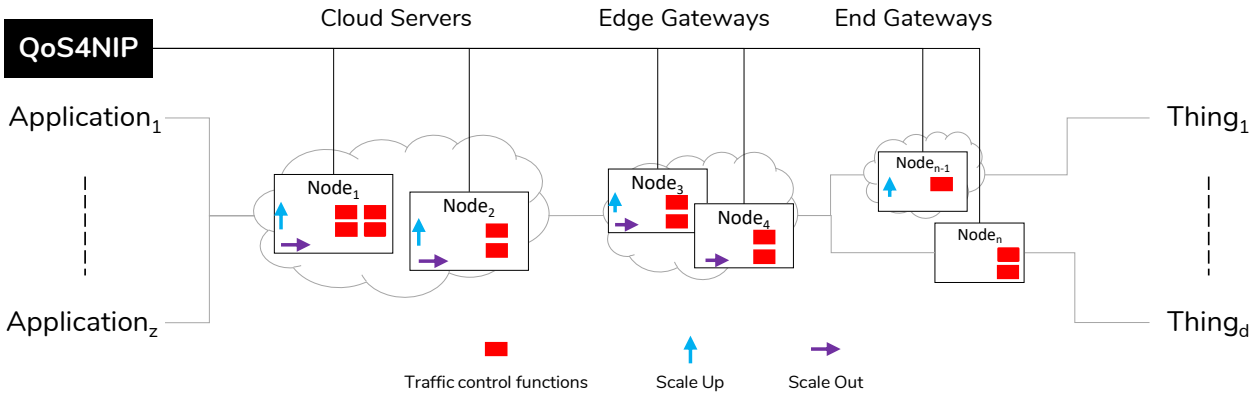


Figure 1: Approach overview over the Cloud-to-Thing continuum.

3.1 The Traffic Control Functions

We handle the NIPs that implement the common reference architectures, such as oneM2M [1]. More specifically, we deal with *stateless* communication between the platform nodes (i.e., Server and Gateways). Such architectural frameworks allow TCFs to be inserted in the platform nodes without disturbing the supported IoT applications. Based on these features, we consider the TCFs proposed at the network level by Carpenter et al. in [27], which we adapt to the specifics of the IoT traffic context. We manage the QoS in a NIP by implementing and distributing on-the-fly the adequate TCFs on the NIP's nodes. We consider that NIPs' nodes in the Cloud Server and Edge Gateways have nested virtualization capabilities for hosting VNF in VM (e.g., running Docker over Amazon EC2 VMs) [28]. Let us remark here that our approach cannot be applied for all platforms, typically multimedia streaming platforms, because of the stateful aspects of End-to-End protocols (i.e., Real-time Transport Protocol (RTP) and Real-time Streaming Protocol (RTSP)) widely used in this context.

3.2 The ANFs packaging solution

To fully enable the deployment of TCFs over the C2TC, we consider the solution explored in [29,30], leading to package NFs into software components that can be deployed on-the-fly on NIPs' nodes. We then distinguish, in Fig 2, two types of NFs. The first type consists of NFs hosted inside virtual containers (VMs or OS-level containers like Docker). This type of function is commonly called VNF [2]. The second type consists of NFs hosted inside a program as a software component. We call them Application NFs (ANF) in the sequel.

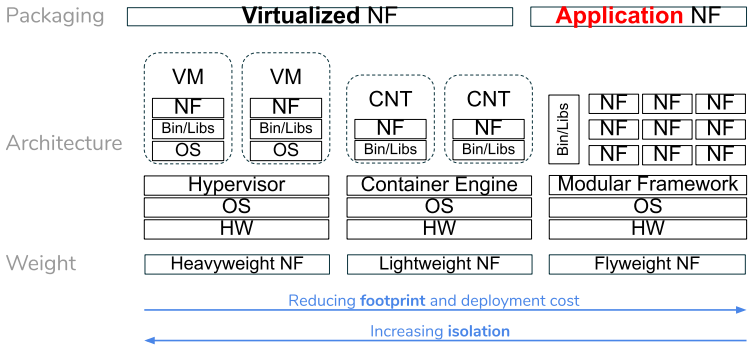


Figure 2: Network Functions Instances.

The ANF concept does not give the same isolation as the VNF concept. Isolation is one of the inherent features highlighted in the existing NFV platforms. Isolation allows the NFs to run on the same hardware and not interfere with each other from two standpoints [31]: security and performance. OS-level containers and VMs are the two virtualization techniques commonly used to provide isolation. On the one hand, different studies, such as [15], show that these virtualization techniques generally induce a high usage of resources. On the other hand, the IoT End Gateways, located close to Things, are generally small devices with limited processing, storage, and connectivity capabilities. For example, an IoT Gateway, such as a *Dell Edge Gateway 5000*, has an Intel Atom processor clocked at 1.33 GHz with only 2 GB of memory. Deploying NFs as VMs or Os-level Containers do not fit NIPs' heterogeneity of resource capacities. Indeed, deploying a VNF as a standard Linux VM requires a minimum[6] of 256 Megabytes RAM, a 300 MHz x86 processor, and 1.5 GB of disk space. Deploying this VNF as a container requires a minimum[7] of 29 Megabytes of disk space. This requirement sharply limits the amount of VNFs that can be deployed on a node and drastically reduces the number of compliant nodes. For most IoT End Gateways, it is difficult to host multiple instances of such VNFs. Moreover, ANF adapts the NF packaging to the constrained deployment context using specific solutions for each chosen programming and deployment environment. The runtime environment "ANF-host" executes on-the-fly ANFs written in a specific programming language (Java in our case). Some of the characteristics of such a runtime environment are:

- an ANF is held and versioned in a code repository;
- ANF dependencies are explicitly defined;
- an ANF can be deployed into development, staging, or production environments without changes;
- an ANF configuration is stored in the environment, typically through environment variables;

[6] https://help.ubuntu.com/community/Installation/SystemRequirements
[7] https://hub.docker.com/r/_/ubuntu/

- backing services, such as data stores, message queues, and memory caches, are accessed through a network, and no distinction is made between local or third-party services;
- an ANF is stateless, and therefore, enables easy scale-out.

All these characteristics allow elementary ANFs to be chained, the same way as VNFs, to provide complex services commonly named SFC (Service Function Chain). For example, when using an OSGi-based modular platform, the ANF is uploaded on the ANF-host through a specific protocol, often HTTP. The OSGi specifications assume an architecture to remotely manage the OSGi framework components relying on a Management Agent (MA). The MA receives, verifies, installs, and configures the ANF according to a "Manifest" file that is similar to the VNFD defined by ETSI-NFV. The method to deploy multiple ANFs (ANFs SFC) implements the "Pre-Calculated Deployment" specified by OSGi [32]. A pre-calculated deployment process is initiated using one of the OSGi subsystem service's install methods. In this case, ANFs SFC (OSGi-based) is an OSGi subsystem deployed as an OSGi Subsystem Archive (.esa) file. An OSGi subsystem comprises resources, including OSGi bundles (ANFs).

3.3 The Joint Optimization of the Scaling Action and the TCFs Deployment

Fig 1 illustrates the overall approach. In this figure, the IoT applications run on top of the NIP (e.g., on a Cloud Server or a User Device). The presence of a square (red) indicates the deployment of a particular TCF on a node (NFV-I or ANF host). The up-arrow (blue) and the right-arrow (purple) indicate the execution of a scaling action on a node (scale-up and scale-out, respectively). Scale-out means adding more instances to a NIP' node, and scale-up means adding more resources to a NIP' node. The overall approach relies on the TCFs deployment on the NIPs' nodes and the scaling action execution to sustain the QoS for the IoT applications.

Nevertheless, the IoT application QoS' fulfillment in NIPs can be seen as a natural Multi-Objective optimization problem. This problem raises a set of optimal solutions (known mainly as Pareto-optimal solutions), instead of a single optimal solution. A solution is Pareto-optimal if we cannot improve any of the objectives without degrading the others. Without additional subjective preference, all Pareto-optimal solutions are considered equally "good." Classical optimization methods suggest converting the Multi-Objective optimization problem artificially to a Single-Objective optimization problem. This usually requires the repetitive use of an algorithm to find multiple Pareto-optimal solutions. On some occasions, such usage does not even guarantee to find Pareto-optimal solutions. In contrast, the population evolution approach of Evolutionary Algorithms (EA) allows an efficient way to find simultaneously multiple Pareto-optimal solutions in a single run [33]. This is the most popular approach in the literature. We implement this approach in this paper. Additional studies on the Multi-Objective EA can be discovered in [34].

In general, the joint optimization of the TCFs deployment and the scaling action execution is an NP-Hard problem. For this problem, we propose a meta-heuristic based on the Genetic Algorithms (GA) that have been proven to constitute an efficient method to provide suitable near-optimal solutions in a short amount of time (see Section 5).

4 Network Functions for TCFs in NIPs

The traffic control mechanisms proposed at the IP level by Carpenter et al. in [27] inspired the proposed functions. [27] introduces DiffServ, an architecture based on a simple model within which the IP traffic that arrives in the network gets assigned to a class of behavior. Each class is uniquely identified by a "Tag" in the IP packets. All the intermediate routers process packets follow the behavior associated with their "Tag." For instance, 80% of the bandwidth of a router belongs to packets tagged A and 20% to those tagged B.

A small number of functions can be composed to differentiate the level of service provided to the IoT applications according to their QoS requirements. The traditional functions (e.i. Classifier, Marker, Dropper, Shaper) are split up simply and deployed when needed. For example, we can deploy a dropping function without the shaping function (avoiding its overhead) and vice versa. We added to these functions a Scheduler and a Redirector. The Classifier and Marker were merged into a new Classifier capable of marking IoT traffic. We package these functions in NFs (ANF and VNF), deploy, and configure them on-the-fly on the targeted NIPs' nodes. In these functions, traffic is composed of one or several messages that cross the NIP's nodes (Server, Gateways); and a traffic profile specifies the temporal properties, such as the rate and the burst size of the traffic. It provides the rules for determining whether a message is in or out-of-profile.

This section presents a) an overview of the TCFs implemented as ANF and VNF to sustain the QoS level to the IoT applications; b) performance evaluations of the VNF and ANF concepts.

4.1 Traffic Control Functions Overview

In the remainder of this paper, a message arriving at a function is denoted r; C denotes the set of considered types of services (or traffic classes), and P denotes the set of traffic priorities. We also consider D, a hash table where the keys range over C, and the values range over P. Below, we explain each of the considered functions, and we propose the algorithms implementing them on NIPs as TCFs.

Classifier. This function is used to "distinguish" the incoming traffic for further processing. The Classifier allows identifying the IoT application traffic and the update of its header with the appropriate label. The Classifier identifies the messages based on their headers' content according to a set of predefined rules, typically some combination of source and destination addresses, content-type, protocols, source, and destination ports fields. Algorithm 1 implements the Classifier. Line 2, the algorithm, first tries to identify the class c of the message r. From lines 3 to 7, the Classifier adds a message header with the associated tag when it recognizes a class. The added header is called the Local Service Level (LSL) header. The time complexity of the Classifier (Algorithm 1) is $\mathcal{O}(|C|)$, where $|C|$ denotes the number of elements of the set C. In practice, we may even handle a fixed number of classes making $|C|$ a constant.

Dropper. This function allows discarding messages based on their LSL header. The Dropper discards some or all messages in an IoT application traffic to bring this traffic into compliance with an expected profile. A REST API is used to configure the rejection percentage and the targeted traffic of this function. Algorithm 2 implements the Dropper. Line 2, upon the reception of a message, the Dropper identifies the associated LSL in the message header. Then, from lines 3 to 6, the algorithm calculates the previously rejected percentage for the considered traffic profile. Line 7, the algorithm rejects the message, return *null* when the percentage of the rejected messages is lower than the specified limit in the configured policy. Otherwise, the Dropper forwards the message without any modification. The time complexity of the Dropper (Algorithm 2) is $\mathcal{O}(|C|)$, where $|C|$ denotes the number of elements of the set C. In practice, we may even handle a fixed number of classes making $|C|$ a constant.

Shaper. This function allows delaying the messages of the traffic to make it compliant with a defined traffic profile. The Shaper discards some messages if there is not enough space in the buffer to hold the delayed messages. The Shaper uses the LSL to identify the delay time of a message. A REST API is used to configure the delaying time and the targeted traffic of this function. Algorithm 3 implements the Shaper. Line 2, the algorithm tries to identify the LSL of the message in its LSL header. From lines 3 to 7, the Shaper holds the message for the necessary delay time matching the identified profile. Line 8, after the elapsed delay, the function returns the message without modification. The time complexity of the Shaper (Algorithm 3) is $\mathcal{O}(|C|)$, where $|C|$ denotes the number of elements of the set C. In practice, we may even handle a fixed amount of classes making $|C|$ a constant.

Scheduler. This function enables the management of the incoming message sequence according to their LSL headers. The function serves any message with a high LSL before a message with a low LSL. If two messages have the same LSL, then the function serves according to their enqueued order. A REST API is used to configure the associated traffic priorities in the queue. Algorithm 4 implements the Scheduler. From lines 1 to 12, the first main procedure enqueues the received message in an internal queue. It delivers this message while it moves to the head of the queue. From lines 13 to 16, the second procedure reorders the messages inside the queue according to their LSL. The time complexity of the Scheduler (Algorithm 4) is $\mathcal{O}(|C| + |Q| \log |Q|)$, where $|C|$ denotes the number of elements of the set C and $|Q|$ denotes the length of the queue Q. In practice, we may even handle a fixed number of classes making $|C|$ a constant, and then, the time complexity is $\mathcal{O}(|Q| \log |Q|)$.

Redirector. This function enables the interception and the forwarding of traffic messages towards different targets. The routing scheme (at the platform level) is affected by this function since we are using an oneM2M-based [1] NIP where the routing is at IoT application-level routing (level 6 of OSI layering). This modification is completely transparent to the IoT application. A REST API is used to configure the new destination and the targeted traffic for this function. Algorithm 5 implements the Redirector. Line 2, the Redirector, identifies the LSL of the message according to its LSL header. From lines 3 to 6, it changes the message's destination according to the corresponding identified LSL. Line 7 it sends the message to its new destination without an LSL and additional modifications. The time complexity of the Redirector (Algorithm 5) is $\mathcal{O}(|C|)$, where $|C|$ denotes the number of elements of the set C. In practice, we may even handle a fixed number of classes making $|C|$ a constant.

Algorithm 1: Classifier Network Function

```
Input: r; D = (C, P)
Output: r
1  begin
2      c ⟵ FindClass(r)
3      if c ≠ {} then
4          for k ∈ D[C] do
5              if k = c then
6                  p ⟵ D[k]
7          r ⟵ AddMark(r, p)
8      return r
```

Algorithm 2: Dropper Network Function

```
Input: r; D = (C, P);
Output: r or null
1  begin
2      p ⟵ GetMessagePriority(r)
3      if p ≠ {} then
4          for k ∈ D[C]) do
5              if k = p then
6                  p_past ⟵ GetRejectionPercentage(k)
7                  if p_past < D[k] then
8                      return null
9                  UpdateRejectionPercentage(k)
10     return r
```

Algorithm 3: Shaper Network Function

```
Input: r; D = (C, P);
Output: r
1  begin
2      p ⟵ GetMessagePriority(r)
3      if p ≠ {} then
4          for k ∈ D[C]) do
5              if k = p then
6                  d ⟵ GetDelay(k)
7                  Wait(d)
8      return r
```

Algorithm 4: Scheduler Network Function

```
Input: r; D = (C, P); Queue Q;
Output: r
1  Algorithm Priority-based Processing()
2      begin
3          p ⟵ GetMessagePriority(r)
4          if p ≠ {} then
5              for k ∈ D[C]) do
6                  if k = p then
7                      Q.push(k, r)
8                      while r ¬ Q.peek() do
9                          Wait()
10                     r ⟵ Q.pull()
11         return r
12 Procedure PriortySortQ()
13     while true do
14         if Q ≠ {} then
15             Timsort(Q)
```

Algorithm 5: Redirector Network Function

```
Input: r; D = (C, P);
Output: r
1  begin
2      p ⟵ GetMessagePriority(r)
3      if p ≠ {} then
4          for k ∈ D[C]) do
5              if k = p then
6                  d ⟵ GetRedirectionIP(k)
7                  r ⟵ SendTo(r, d)
8      return r
```

4.2 Evaluation of the TCFs packaging (VNF and ANF)

In this section, we evaluate the deployment time of the TCFs implemented as VNF and ANF. Then, we study the effects of the traffic arrival rates on the processing time and the resource usage (CPU and RAM) required for executing the TCFs. The goal is to get quantitative characteristics associated with the different packaging (ANF and VNF) of the TCFs.

The details of the TCFs implementation are provided in the Appendix.

Experimental context. The presented performance measurements allow assessing the deployment time, denoted d_t, of the TCFs: as ANF in the considered ANF-host (i.e., IoT Gateway); and as VNF in the considered NFV-I nodes. In order not to bias the tests by an additional upload time related to network conditions, the TCFs are supposed to be already present in the hosting system as Docker Images for VNFs, and JARs files for ANFs. To collect performance metrics, we implemented monitoring tools based on Java Management Extensions (JMX) technology. In each TCF, we created MBeans objects for processing time, CPU, and RAM remote monitoring.

We characterize the processing time, denoted p_t, associated with each function under the effects of request arrivals. Let a session $\mathcal{S} = (r_1, r_2, \ldots, r_n)$ be a sequence of n requests for resource r_i coming from the same IoT application, and let $t_r(r_i)$ and $t_s(r_i)$, respectively, be the time that resource r_i was requested and the time that resource r_i was served, respectively. The processing time for request r_i in session $\mathcal{S}$ is:

$$p_t(r_i) = t_s(r_i) - t_r(r_i) \tag{1}$$

According to [35], the Poisson distribution for modeling the traffic of an IoT application to the Cloud is a good approximation for the scalability analysis. Thus, to simplify, we assume that the arrivals of the IoT traffic in a session follow a Poisson distribution. An event (request arrival) can occur k times (0 to n) in a given

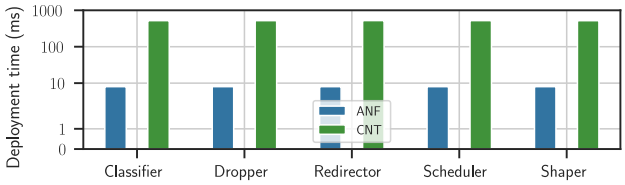


Figure 3: Traffic Control Functions deployment time.

interval. The probability P of observing k events in an interval is given by Equation:

$$P(k) = e^{-\lambda}\frac{\lambda^k}{k!} \tag{2}$$

where k is the number of times an event occurs in an interval and takes values $0, 1, 2, \ldots$.

The experimental testbed consists of three host machines: one traffic generator equipped with two CPU and 4 GB RAM, one NFV-I equipped with four CPU and 16 GB RAM, and one ANF-host fitted with one CPU and 4 GB RAM. All the CPUs are CPU Intel Core i7-7500U clocked at 2.70 GHz. The traffic generator produces the IoT traffics according to a Poisson distribution with a request arrival rate of $\lambda \in [1, 50, 100, 150]$ req/s (request size = 1 Mb). The NFV-I is composed of all hardware and software components that build up the environment in which VNFs are deployed and managed using the OpenBaton [36] platform. The ANF-host is running an OSGi-based [32] program that can deploy ANFs. The three host machines run with Ubuntu 16.04. The template (*size*) of a VNF/ANF is 1 CPU and 4 Gigabytes RAM. In these experiments, a message is an HTTP request or an HTTP response. The considered NIP is the Eclipse open-source OM2M [37] that implements the standard oneM2M [1].

The conducted experiments address the following questions:

(a) How does the TCF type impact the deployment time;

(b) How does λ in Equation (2) impact the processing time defined in Equation (1);

(c) How does λ in Equation (2) impact the CPU and RAM usage;

(d) How does the CPU and RAM saturation impact the TCF performance.

Performance analysis. In the first experiment, we answer the question "(a)" by investigating the *TCF deployment time*. We examine the relationship between the TCF type (ANF and VNF) and their deployment time. Fig. 3 shows the results in a logarithmic scale. The deployment time of an ANF with an average weight of 15 Kbits is ≈ 8 ms; the deployment time of a VNF having an average weight of 200 Megabytes is ≈ 520 ms.

The second experiment investigates the *TCF processing time* to answer the question "(b)." We analyze the relationship between the request arrival rate λ and the processing time p_t. We start with each implementation (ANF and VNF) of each TCF facing a session $\mathcal{S}$ of 3000 requests and a $\lambda = 1$. Then, repeatedly, with the same session $\mathcal{S}$ of 3000 requests, we increase λ first to 50, then to 100, and finally to 150. The results (shown in Fig. 4a and Fig. 4b) confirm the expected behavior: the increase of λ leads to the increase of the processing time. For instance, in Fig. 4a, with $\lambda = 1$, we have a $p_t(min) = 0$ ms, $p_t(median) = 2$ ms, $p_t(max) = 50$ ms for the Dropper Network Function processing time. However, the cumulative distribution function (CDF) of the same TCF facing the same λ differs depending on its type (ANF or VNF). In Fig. 4a and Fig. 4b, the Classifier processing time represents the insertion of the tag (LSL). For instance, in Fig. 4a, it takes 2 ms to insert a tag for almost 75% of the requests when we handle 1 req/s. Additionally, we handle tags only internally inside an infrastructure node of the NFV network topology (NFV-I). The Classifier calculates the tag in each node according to the content of the message headers. The tag is not transmitted outside of the NFV-I node entities, and no transmission overhead is then induced by the message exchange between the NFV-I nodes, which is the most significant part of the communication traffic. The same applies to ANF-host.

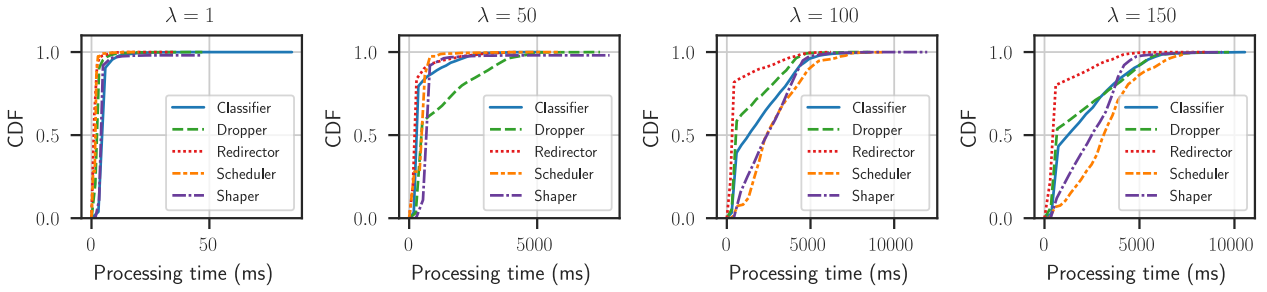
In the third experiment, we answer the question "(c)" by investigating the *TCF resource usage*. We audit the relationship between the request arrival rate λ and the resource usage (CPU and RAM). We start with each implementation (ANF and VNF) of each TCF facing a session $\mathcal{S}$ of 3000 requests and a $\lambda = 1$. Then, we repeat, with the same session $\mathcal{S}$ of 3000 requests, raising λ first to 50, then to 100, and finally to 150. Using ANF, there is essentially no isolation in the use of resources, so we approximate the ANF resource usage to the whole resource usage of the Java Virtual Machine hosting it, which is the worst situation of resource usage. For each session of every TCF (ANF, VNF), we measure the average usage of CPU and RAM. Fig. 4c and Fig. 4d show these average usages. The results show that the TCFs implemented as VNFs consume more CPU compared to ANFs. However, both (ANFs and VNFs) consume the same amount of RAM.

Saturation effect on TCFs: Here, we answer the question "(d)" by exploring the relationship between the resource saturation (i.e., when CPU and/or RAM are utilized over 90%) and the TCF performance. As shown in Fig. 4a and Fig. 4b, the CPU saturation has an important influence on the VNF TCFs performance. From $\lambda = 50$, we can see that, on the one hand, the CPU is used at $\approx 100\%$, and the p_t reaches 5 seconds (Fig. 4a). On the other hand, the RAM remains slightly congested because the proposed TCFs are almost stateless and computation intensive.

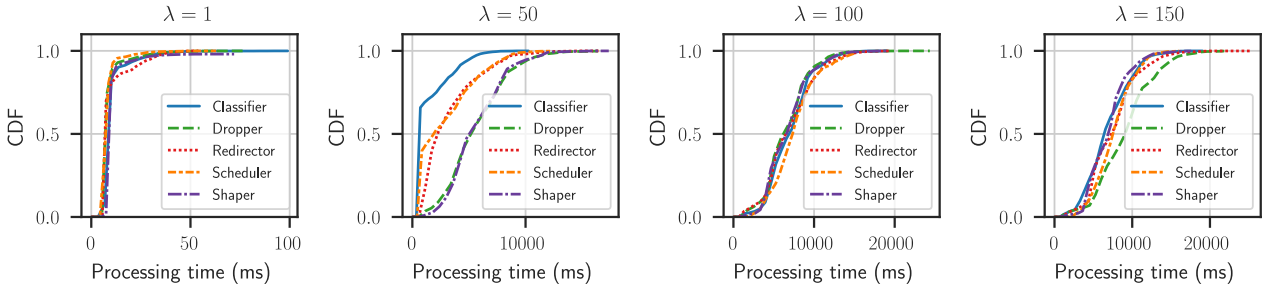
By removing the isolation between NFs, we lose a level of security. However, we get a decrease in the overhead (resource usage, deployment time), a reduced complexity of the hosting nodes, and the increase of the number of hosting nodes. Considering, under some circumstances, the isolation as a non-mandatory functionality for the deployment of NFs, the concept of ANF completes the global toolset that sustains QoS in NIPs. Our approach aims to dynamically deploy the different TCFs presented in this section within the platform, according to resources and requirements changes. Given the task's complexity, several TCFs can be considered, but with varying results. Section 5 presents our contribution, based on a combinatorial optimization heuristics, to decide the best combination of TCFs to deploy appropriately to the current context. Section 6 presents the method of evaluating the performance of our contribution, as well as the results obtained.

5 Design of QoS4NIP Planner

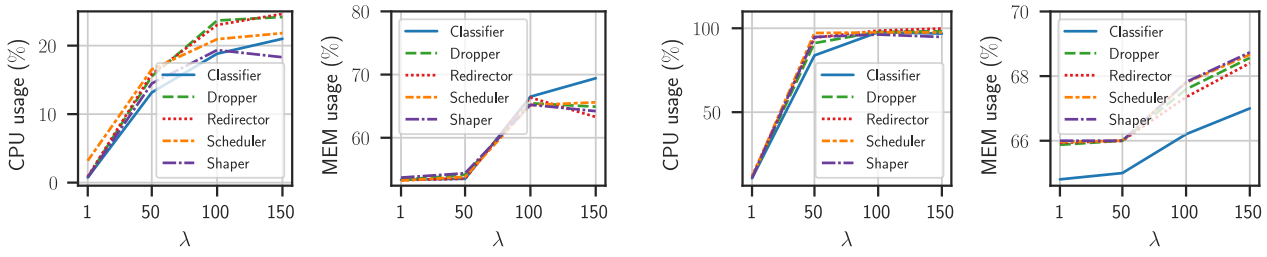
In a real scenario, as described in Section 6, the proposed planner, called QoS4NIP, is located on top of the NIP's monitoring system. This follows the autonomic architecture model of [38]. QoS4NIP is invoked periodically and takes the monitoring information as inputs. The output of QoS4NIP is a reconfiguration plan represented by a binary vector. The configuration enforcement component [38] performs this reconfiguration and takes into account the current configuration. For instance, when the QoS4NIP reconfiguration plan includes deploying a



(a) ANF Processing Time.



(b) VNF Processing Time.



(c) ANF Resource Usage.

(d) VNF Resource Usage.

Figure 4: Traffic Control Functions performances.

Table 2: Notations

Names	Meanings
τ	an IoT application
z	the total number of IoT applications
n	the total number of nodes
L_{Qos_τ}	the Latency required by τ
T_{Qos_τ}	the Throughput required by τ
U_{Qos_τ}	the Unavailability required by τ
L_{E2E_τ}	the End-to-End Latency served to τ
T_{E2E_τ}	the End-to-End Throughput served to τ
U_{E2E_τ}	the End-to-End Unavailability served to τ
L_{i_τ}	the Latency of τ on node i
T_{i_τ}	the Throughput of τ on node i
U_{i_τ}	the Unavailability of τ on node i
ρ_{i_τ}	the monitored Throughput on node i for τ
δ_i	the monitored Latency on node i
m	the total number of scaling actions
p	the total number of TCFs
A_i	the set of scaling actions supported by the node i
F_i	the set of TCFs supported by the node i
f_q	the TCF q benefit
a_c	the scaling actions c benefit
Γ_{i_c}	the cost of scaling action c on node i
cpu_q	the TCF q cpu resource usage
ram_q	the TCF q ram resource usage
η_i	the sum of the benefits induced by all the supported TCFs and the scaling actions on the node i
ζ_i	the sum of the benefits of the Throughput induced by all the supported scaling actions on the node i
ω_i	the weighting factor of the Scheduler on the Throughput on the node i
ϵ_i	the loss factor of the Dropper on the Unavailability on the node i
λ_i	the request arrival rate on node i
$Cost_{E2E}$	the End-to-End Cost of the scaling action
$Cost_i$	the cost of all the scaling action execution on node i
RU_{E2E}	the End-to-End Resource usage of TCFs
RU_i	the resource usage of TCFs on node i
$H_{i_{cpu}}$	the node i cpu usage
$H_{i_{ram}}$	the node i ram usage
β_i	the node i cpu usage with Scaling action execution and TCFs deployment
γ_i	the node i ram usage with Scaling action execution and TCFs deployment
X_θ	the binary vector describing the describe the application of TCFs or scaling actions to the NIPs' nodes
x_i^j	a binary row of X_θ
T_θ	the integer matrix describing the genes' additional information
t_i^j	a integer row of T_θ
P_t	a Pareto front
C_p	the crossover probability
M_p	the mutation probability
N	the population size
l	the chromosome length
T	the maximum number of generations

given TCF on a particular NIP's node, and if this given TCF is already deployed, nothing happens. Otherwise, the TCF will be deployed. The same applies when the QoS4NIP reconfiguration plan does not include the deployment of a given TCF on a particular NIP's node (this TCF will be removed). QoS4NIP handles a scaled out/up node, virtually, as a *unique* node with i) resized resource in case of scaling-up, and ii) combined resources in case of scaling-out.

In this section, we describe the design of QoS4NIP based on the considerations mentioned above. QoS4NIP considers the different trade-offs for the cognitive management of QoS in NIPs. In Section 5.1, we describe the considered system model. In Section 5.2, we formulate the QoS model of IoT applications (Latency, Throughput, and Availability), the scaling actions cost model, and the TCFs deployment resource usage model. As stated in Section 3.3, we formulate in Section 5.3 a multi-objective optimization problem for efficient planning. We propose in Section 5.4 a modelization for the problem resolution (GA-based Constrained Optimization Model). To solve the multi-objective optimization problem, we explore in Section 5.5 the evolutionary strategies and the Pareto front. Finally, we present in Section 5.6, the QoS4NIP planner algorithm. For convenience, the used notations are listed in Table 2.

5.1 System Model

Fig. 1 depicts the system model used by the multi-objective optimization algorithm presented in this paper. Let the NIP be composed of a set of n TCF (VNF or ANF) hosting nodes that are already provisioned and are parts of the infrastructure. Let consider that each TCF or scaling action is associated with a benefit (estimated a priori). This could be justified; for example, using the classical response time model $R = S/(1-U)$, where S is the node service time, U is the node utilization. The execution of a scaling action decreases U and therefore decreases R. For simplicity, we call this variation (R/R_{scaled}) the "benefit" of the scaling action. This benefit, expressed as a percentage (%), describes how the scaling action affects the QoS on the hosting node. The same applies to the TCFs. For instance, a benefit of 25% means that the TCF or the scaling action reduces the targeted IoT application Latency by 25% (to the detriment of other applications that are not targeted).

The *joint optimization problem* is to find the relevant TCFs to deploy (or remove) on every node of this set and the scaling actions to execute, while optimizing the overall E2E QoS (i.e., E2E Latency, Throughput, and Unavailability).

5.2 QoS, Cost and Resource Usage Models

Given a set of z IoT applications, we compute for each IoT application τ: the E2E Latency (denoted L_{E2E_τ}), the E2E Throughput (denoted T_{E2E_τ}), the E2E Unavailability (denoted U_{E2E_τ}). We also compute the resource usage associated with the deployment of the TCFs (RU_{E2E}), and the cost associated with the execution of the scaling actions ($Cost_{E2E}$).

E2E Latency model. As per [39], we can easily calculate the E2E Latency as the sum of all the local Latencies for IoT application τ on the n nodes.

$$L_{E2E_\tau} = \sum_{i=1}^{n} L_{i_\tau} \tag{3}$$

In Equation 3, we assume a zero-latency for the IoT application τ if the benefit η_i (i.e. the sum of the benefits induced by all the supported TCFs and the scaling actions on the node i) is greater than 100. Otherwise, the Latency on the node i is $(1-\eta_i\%)$ of the monitored Latency.

$$L_{i_\tau} = \begin{cases} 0 & \text{if } \eta_i \geq 100 \\ \delta_i \times (1-\eta_i\%) & \text{else} \end{cases} \tag{4}$$
$$\text{with } \eta_i = \textstyle\sum_{q=0}^{p} f_q + \sum_{c=0}^{m} a_c$$
$$q \in F_i \text{ and } c \in A_i$$

E2E Throughput model. The E2E Throughput is the minimum of all the Throughputs crossed by the IoT application τ.

$$T_{E2E_\tau} = min(T_{1_\tau} \dots T_{n_\tau}) \tag{5}$$

Where we assume that ζ_i is the sum of the benefits to the Throughput induced by all the supported scaling actions on the node i. The Throughput on the node i is then the monitored Throughput added to ζ_i, if no Scheduler is deployed or if node i does not support Scheduler deployment. When a Scheduler is on the node i, then the Throughput is $\omega_i\%$ of the monitored Throughput added to ζ_i.

$$T_{i_\tau} = \begin{cases} \rho_{i_\tau} - \zeta_i + 1 & \text{if } \nexists \text{ scheduler} \vee \text{scheduler} \notin F_i \\ (\rho_{i_\tau} - \zeta_i + 1) \times \omega_i & \text{else} \end{cases} \tag{6}$$
$$\text{with } \omega_i = 1 + \tfrac{1}{100} f_{scheduler} \text{ and } \zeta_i = \tfrac{1}{100} \textstyle\sum_{c=0}^{m} a_c$$

E2E Unavailability model. The E2E Unavailability is the sum of the Unavailability of the n nodes for the IoT application τ.

$$U_{E2E_\tau} = \sum_{i=1}^{n} U_{i_\tau} \tag{7}$$

Where we assume that the Unavailability of node i is zero if there is no Dropper deployed or if node i does not support the Dropper deployment. If there is a Dropper deployed on the first node (0), then the Unavailability is the rejection percentage associated with the IoT application τ. Otherwise, the Unavailability is the remaining availability multiplied by the rejection % associated with the IoT application τ.

$$U_{i_\tau} = \begin{cases} 0 & \text{if } \nexists \text{ dropper} \vee \text{dropper} \notin F_i \\ f_{dropper} & \text{if } i = 0 \\ (1-\epsilon_i) \times f_{dropper} & \text{if } i > 0 \end{cases} \tag{8}$$

$$\text{with } \epsilon_i = 1 - \frac{1}{100}\sum_{i'=0}^{i-1} U_{i'_\tau}$$

Scaling action execution's E2E Cost model. The $Cost_{E2E}$ is the sum of all the costs associated with the scaling actions execution on the node i ($Cost_i$).

$$Cost_{E2E} = \sum_{i=1}^{n} Cost_i \tag{9}$$

where

$$Cost_i = \begin{cases} \sum_{c=0}^{m} \Gamma_{i_c} & \text{if node } i \text{ support all} \\ & \text{the scaling actions} \\ \infty & \text{else} \end{cases} \tag{10}$$

TCFs deployment's E2E resource usage model. The RU_{E2E}, is the sum of all the resource usage associated with the deployment of TCFs on the nodes.

$$RU_{E2E} = \sum_{i=1}^{n} RU_i \tag{11}$$

Where

$$RU_i = \begin{cases} \sum_{q=0}^{p}(cpu_q(\lambda_i) + ram_q(\lambda_i)) & \text{if} \beta_i\% \leq 1 \\ & \wedge \gamma_i\% \leq 1 \\ \infty & \text{else} \end{cases} \tag{12}$$

$$\text{with } \beta_i = (\sum_{q=0}^{p} cpu_q(\lambda_i) + H_{i_{cpu})} \times \sum_{c=0}^{m} a_c$$

$$\text{and } \gamma_i = (\sum_{q=0}^{p} ram_q(\lambda_i) + H_{i_{ram})} \times \sum_{c=0}^{m} a_c$$

The CPU and RAM usage of the TCF depend on the request arrival rate λ on the node i.

5.3 Multi-Objective Problem Formulation

We formulate in this section a multi-objective optimization problem for efficient planning of the TCFs (proposed in Section 4) and scaling actions execution in the multi-constraint NIP set-up. Our goal in the formulated problem is to minimize the **ratio** between the IoT application's QoS requirement and the QoS provided (E2E Latency, E2E Availability, and E2E Throughput) by the NIP. The k-objectives problem is formulated as:

$$\begin{aligned} \text{minimize} \quad & F = l_1, \ldots, l_z, t_1, \ldots, t_z, u_1, \ldots, u_z \\ \text{subject to} \quad & l_\tau \leq 1, \forall \tau \in [1, ..., z], \\ & t_\tau \leq 1, \forall \tau \in [1, ..., z], \\ & u_\tau \leq 1, \forall \tau \in [1, ..., z] \end{aligned} \tag{13}$$

Where we have

$$l_\tau = \frac{L_{E2E_\tau}}{L_{Qos_\tau}} \tag{14}$$

$$t_\tau = \frac{T_{Qos_\tau}}{T_{E2E_\tau}} \tag{15}$$

$$u_\tau = \frac{U_{E2E_\tau}}{U_{Qos_\tau}} \tag{16}$$

5.4 GA-based Constrained Optimization Model

In this section, we define the "individuals" structure (chromosome). A chromosome is a solution that combines the execution of scaling actions and TCFs deployment to sustain QoS. Additionally, we consider the following genetic operators: mutation and crossover.

Genotype. The solutions are represented in a way that they can be easily understood and manipulated. We define a chromosome as a binary vector X_θ to describe the application of TCFs or scaling actions to the

NIP's nodes (See Fig. 5a). Each X_θ is associated with an integer matrix T_θ that contains the TCF or scaling actions' additional information.

$$X_\theta = [x_1^1, \ldots, x_1^{p+m}, \ldots, x_n^1 \ldots x_n^{p+m}] \tag{17}$$

Particularly, each decision variable $x_i^j = 1$ if and only if the j^{th} TCF or scaling action is applied to NIP's node i. Each decision variable x_i^j is associated with a configuration vector named t_i^j. The vector t_i^j is represented in Fig. 5b and contains the following information:

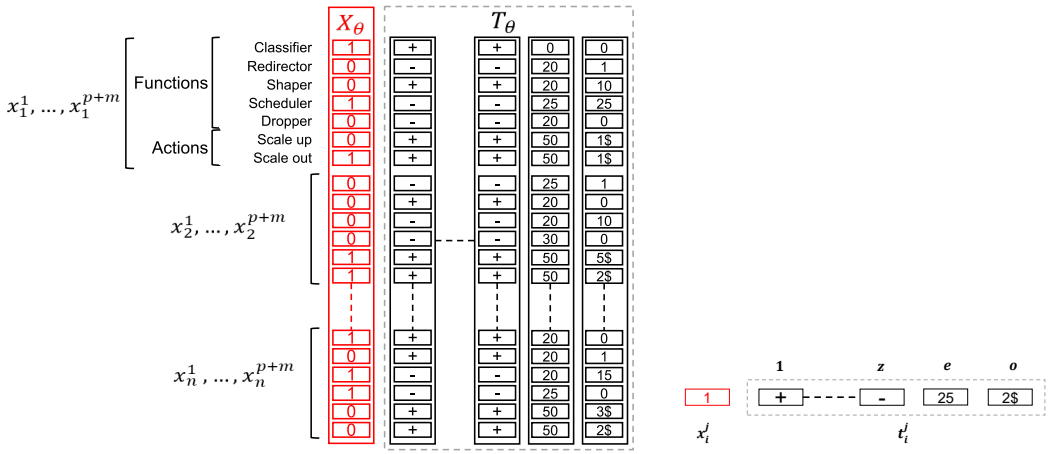
- $t_i^j[1 \ldots z]$: denotes the decision variable x_i^j effect on all IoT applications.

$$t_i^j(\tau) = \begin{cases} +1 & \text{if } x_i^j \text{ improves the QoS of } \tau \\ -1 & \text{else} \end{cases} \tag{18}$$

- $t_i^j[e]$: denotes the proportion of the decision variable x_i^j effect. For instance, $t_i^j[e] = 25$ means that the decision variable x_i^j can reduce (if $t_i^j[\tau] > 0$) or increase (if $t_i^j[\tau] < 0$) the IoT application τ Latency by 25%.
- $t_i^j[o]$: denotes information to each the decision variable x_i^j.
  - In the Redirector gene, $t_i^j[o]$ denotes the number of hops for the priority IoT application.
  - In the Shaper gene, $t_i^j[o]$ denotes the delay time for the priority IoT application.
  - In the Scheduler gene, $t_i^j[o]$ denotes the scheduling rate for the priority IoT application.
  - In the Scale-up and Scale-out genes, $t_i^j[o]$ denotes the cost in USD.

Genetic operators. We consider the classic operators that are enough to create and maintain the genetic diversity, by combining existing solutions into new solutions, and to select between solutions:

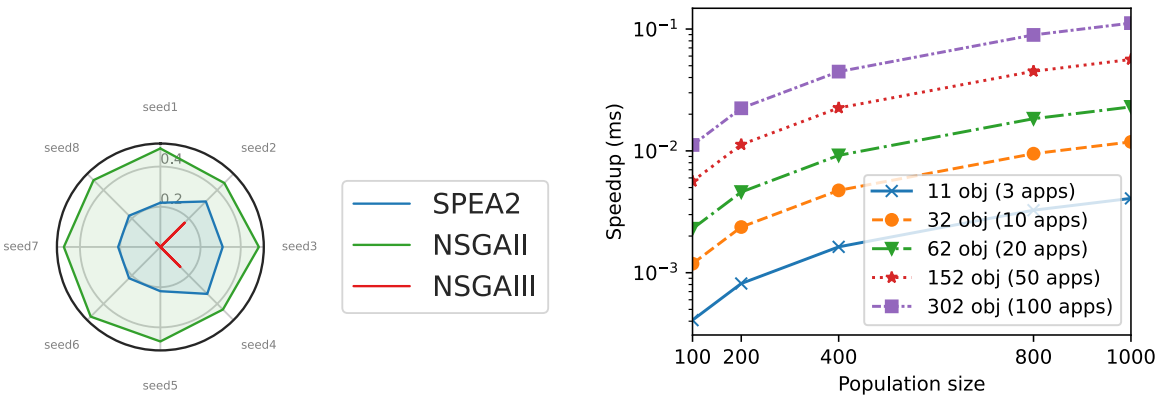
- Bit-flip – acts independently on each bit in a solution and changes the value of the bit (0 to 1 and vice versa) with probability M_p, where M_p is a parameter of the operator. The most commonly prescribed value for this parameter is $M_p = 1/l$.
- Tournament Selector (Selection) – selects an individual from a population of individuals by running several "tournaments" among a few individuals randomly chosen from the population. This operator selects the winner of each tournament (the one with the best fitness) for crossover.
- Half Uniform Crossover (HUX) – swaps the half of the non-matching bits of two solutions according to a probability C_p. For this purpose, HUX first calculates the number of different bits (Hamming distance) between the parents. Half of this number is the number of bits exchanged between parents to form the two children.



(a) Chromosome X_θ.

(b) Illustration of x_i^j and its associated t_i^j.

Figure 5: Genotype Representation.



(a) Hyper-volume measure on the formulated problem with $C_p = 100\%$; $M_p = 100\%$; N = 200, $l = 28$.

(b) Speedup achieved by the NSGAII-based evolutionary strategy on the formulated problem for 3, 10, 20, 50 and 100 IoT applications (on a Processor Intel (R) Core (TM) i7-7500U CPU @2.70GHz).

Figure 6: Evolutionary strategy on the formulated problem.

5.5 Evolutionary Strategies and Pareto Front analysis

The proposed model above is the starting point in the implementation of a Genetic Algorithm to optimize the QoS parameters of IoT applications (Latencies, Throughputs, Availabilities), resource usage of TCFs, and cost of scaling actions. In this section, firstly, we present the adopted evolutionary strategy to compare individuals. Secondly, we present a discussion on the choice of the solution in the Pareto front to apply.

Evolutionary strategy. We adopted the evolution strategy for QoS4NIP planner based on the Hyper-volume calculated from Pareto fronts found by the main algorithms in the literature, and that are compatible with the formulated problem. We consider the Non-dominated Sorting Genetic Algorithm II (NSGAII [33]), III (NSGAIII [40]), and the Strength Pareto Evolutionary Algorithm 2 (SPEA2 [41]). The Hyper-volume indicator measures the volume of the dominated portion of the objective space. It is of exceptional interest, as it possesses a highly desirable feature called strict Pareto compliance. This feature means that whenever one approximation completely dominates another approximation, the Hyper-volume of the former will be higher than the Hyper-volume of the latter.

The largest Hyper-volume was obtained by NSGAII, as shown in Fig. 6a. The outperformance of NSGAII on NSGAIII is explainable since our problem is of type Knapsack Problems (KP). As clearly demonstrated in [42], on multi-objective KP, NSGAII outperformed NSGAIII. NSGAII will be used for validation purposes in the rest of this paper. The reader may see [33] for further details about the NSGAII algorithm.

Discussion on the choice of the applied solution. As earlier stated, the presence of multiple objectives in a problem, in principle, gives rise to a set of optimal solutions. None of these Pareto-optimal solutions can be considered better than the others in the absence of additional information. In our context, once the GA finds a Pareto front, a choice must be made to apply a unique solution to the NIP. We recommend three methods of selection.

The first method is the *Random Selection*, which consists of choosing a solution randomly from the Pareto front. This method is a proper selection since each solution, X_θ, of the Pareto front has the same probability of being applied to the NIP.

The second method is the *QoS objectives-based Selection*. This method consists of selecting a solution to apply to the NIP based on the ranking or weighting of the QoS. For instance, in the Pareto front, there may be some non-dominated solutions leading to request losses for some IoT applications. The choice will be towards the solution that discards nothing, even if it proposes higher Latencies ($\leq L_{qos}$).

The last method of selection is the *Non-QoS objectives-based Selection*. The selection of the solution to be applied to the NIP is based on Non-QoS criteria, such as the number of scaling actions and TCFs required by the solution (Complexity-based), or the cost and resource usage associated with each solution (Cost-based). The following case study on Connected Vehicles will use this last method (Cost-based).

5.6 The QoS4NIP Planner Algorithm

The general work-flow of the QoS4NIP planner (NSGAII-based) presented in Algorithm 6 is as follows.

From lines 2 to 4, the population is initialized randomly, where every individual's structure is as proposed in Fig. 5a. Then, the fitness value of every solution in the current population is computed using Equations 3, 5, 7 and the monitoring information (cf. Equations 4 and 6). All the individuals of the current population

Algorithm 6: QoS4NIP Planner

Input: N; T
Output: X_θ
1 **begin**
2 Set $t = 0$ Initialize P_0 and set $Q_0 = \emptyset$.
3 **while** $t < T$ **do**
4 Calculate fitness for P_t and assign rant based on Pareto dominance
5 Perform selection on P_t to fill the mating pool
6 Apply crossover and mutation operators to obtain the offspring population Q_t
7 Select the best N non-dominated solution from $P_t \cup Q_t$ by the two-step procedure to form P_{t+1}
8 Set $t = t + 1$
9 Set $j = 0$
10 **while** $j < N$ **do**
11 Calculate and save $RU_{E2E}[j]$ for $P_t[j]$
12 Calculate and save $Cost_{E2E}[j]$ for $P_t[j]$
13 Set $j = j + 1$
14 $indexes \longleftarrow \arg\min_{j=1...N}\{Cost_{E2E}(P_t[j])\}$
15 $index \longleftarrow \arg\min_{j \in indexes}\{RU_{E2E}(P_t[j])\}$
16 $X_\theta \longleftarrow P_t[index]$
17 **return** X_θ

with penalties values are discarded. Once the fitness is assigned, the population is sorted according to the non-domination individual. Line 5, the Tournament selector is applied to the entire population to determine the fittest individuals of the current population, which will be placed into the mating pool. Line 6, new solutions, called offspring, are generated by applying Bit-Flip Mutation and Half Uniform Crossover to the mating pool. Line 7, based on the values provided by the ranking scheme, the best individuals from the combination of the current population P_t, and the offspring pool Q_t, are detected. Those with a lower value (min) or higher crowding distance are saved in the next population P_{t+1}. The crowding distance mechanism is used to preserve the diversity of solutions. It estimates the volume of the hyper-rectangle defined by two nearest neighbors [33]. If some candidate solutions are of the same rank, and not all of them can enter the next population, the less crowded individuals from a given rank are selected to fit the future population. From lines 9 to 13, the Pareto Front's scaling action cost and resource usage are calculated. From lines 14 to 17, using the selection method described in Section 5.5, the cheapest solution (X_θ) is returned. The heuristic time complexity is $\mathcal{O}(MN^2)$, where M is the number of objectives, and N is the population size. The plots in Fig. 6b. have been drawn in logarithmic scales. They show the speedup in ms as a function of population size.

6 Evaluations in a Connected Vehicles Case Study

Much of the data required by Connected Vehicles can be transferred using short-distance communications. However, numerous use cases depend on the information that is not obtainable within proximity. For these longer communication paths, the cellular network could be a potential solution for communication, both between vehicles and from the vehicles to the network itself, so-called vehicle-to-network (V2N) communication. As shown in Fig 7, we consider three realistic V2N IoT applications with different QoS requirements [43]. We carried out simulations to evaluate the effectiveness of the proposed approach against others.

6.1 Compared Schemes

The relative performance comparison of the proposed scheme (QoS4NIP) has been carried out against four other schemes. The first is the standard First-Come-First-Served (FCFS)-based approach. Unlike the proposed scheme, this approach does not give any particular emphasis to maintain QoS requirements. The second is the autoscaling scheme. To not bias the results, we compared our scheme with the autoscaling approach, without considering a particular implementation in the literature, while trying to show its limits. The compared autoscaling scheme is obtained by tuning our planning algorithm to use the scaling actions only and switching off VNF and ANF deployment.

In our comparison, we also want to distinguish the costs induced by VNF usage versus ANF usage, independently of autoscaling. For this purpose, we implemented the two other schemes as two variants of QoS4NIP, wherein one variant only deploys VNFs, while the latter, additionally, considers using ANFs on the IoT End Gateways. In the following, the FCFS, the autoscaling scheme, and the considered variants of QoS4NIP are referred to as FCFS, AS, QoSEF, and QoSEFe, respectively.

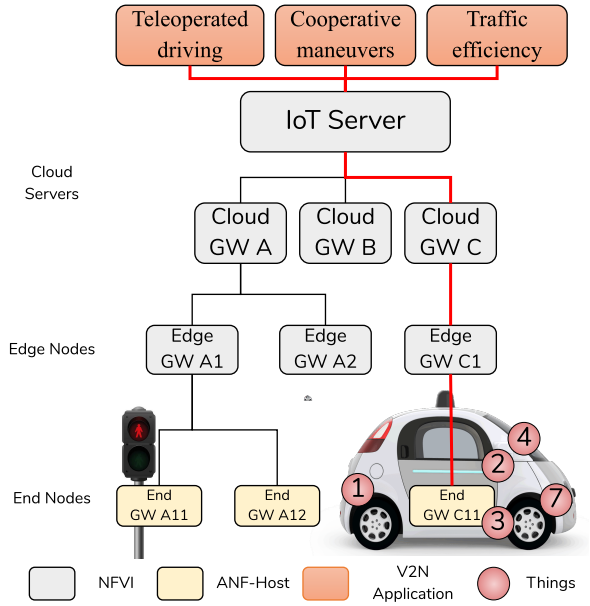


Figure 7: Considered topology for the case study.

6.2 Simulation Setup

Table 3 shows *Teleoperated driving*, *Cooperative maneuvers* and *Traffic efficiency* QoS requirements. All these V2N IoT applications communicate with the actuators and sensors in the vehicle through "IoT Server", Cloud "GW C", Edge "GW C1" and End "GW C11." In each test case, the platform is modeled by a snapshot, s_0, where no TCF is deployed, and no scaling action is executed. We implemented all the compared schemes (AS, QoS4NIP, QoSEF, QoSEFe) in Python using the Multi-Objective Evolutionary Algorithms library Platypus [44].

6.3 Evaluation Parameters

Using the results of our previous work [45, 46], we show the *a priori* benefit of each TCF presented in Table 4. The scaling actions benefits are considered, as shown in the work [47]. For the resource usage parameters (CPU and RAM), we rely on the performance model of ANF and VNF presented in Section 4.

The four considered schemes for comparison (AS, QoSEF, QoSEFe and QoS4NIP) are initialized with the snapshot of the FCFS scheme, s_0 (presented in Table 5), corresponding to a number of objectives = 9, N = 200, C_p = 100%, and M_p = 1; with $n = 4$, $p = 5$, $m = 2$, and $l = 28$ (i.e. $n \times (p + m)$). The resource usage ($H_{i_{ram}}$ and $H_{i_{cpu}}$) on each node (IoT Server, Cloud "GW C", Edge "GW C1", and End "GW C11") is [15-25]%. Scale-up and Scale-out cost per node is fixed to 0.3 USD (corresponding to an "AWS r4.large" price in march 2020).

Table 3: Representative V2N applications. T= Throughput in req/sec (request size = 1Mb); L= Latency in ms; A= Availability in %.

V2N Application	Description	QoS Requirements		
		T	L	A
Teleoperated driving	An external operator drives the vehicle using a live-stream video.	25	20	99
Cooperative maneuvers	A set of vehicles communicating and behaving as a system for performing coordinated actions.	10	100	99
Traffic efficiency	Optimization of traffic parameters (traffic lights, speed limit, etc.).	10	1000	90

Table 4: Benefits parameter settings

	Benefit	Description
Classification	0%	Deploy a classifier on an node
Redirection	0%	Deploy a redirector on an node
Scheduling	35% [45]	Deploy a scheduler on an node
Shaping	35% [45]	Deploy a shaper on an node
Dropping	41% [45]	Deploy a dropper on an node
Scale out	50% [47]	Replicate an node
Scale up	50% [47]	Double resources of an node

Table 5: Initial snapshot s_0 parameter settings

	QoS offered to applications at s_0		
	Teleoperated driving	Cooperative maneuvers	Taffic efficiency
L_i	10 ms	30 ms	80 ms
U_i	0%	0%	0%
T_i	25 req/sec	10 req/sec	10 req/sec

6.4 Evaluation Metrics

The reconfiguration plan, we refer to here, are those proposed by the solutions associated with the different schemes. The evaluation metrics used to assess the proposed approach are defined as follows:

- E2E Actions Cost and E2E Resource Usage: respectively, the End-to-End costs computed from Equation (9) and the End-to-End resource usage to sustain the QoS computed from Equation (11).
- E2E Latency: End-to-End solutions Latency of *Teleoperated driving, Cooperative maneuvers and Traffic efficiency*, computed from Equation (3).
- E2E Availability: End-to-End solutions Availability of *Teleoperated driving, Cooperative maneuvers, and Traffic efficiency* is 1 minus the End-to-End solutions Unavailability (denoted U_{E2E_τ}), computed from Equation (7).
- E2E Throughput: End-to-End solutions Throughput of *Teleoperated driving, Cooperative maneuvers and Traffic efficiency*, computed from Equation (5).

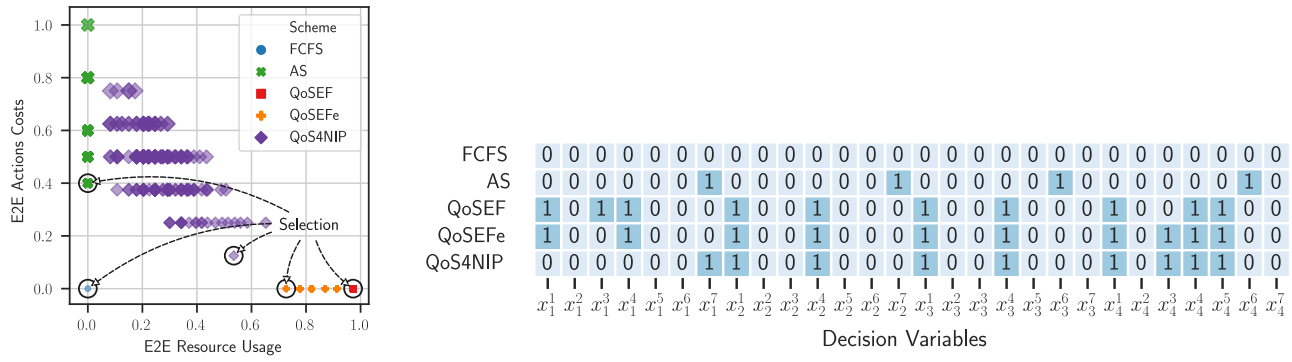
6.5 Observations

This part discusses the results we obtained. We compare the E2E Actions Cost, the E2E Resource Usage, the E2E Latency, the E2E Availability, and the E2E Throughput in the FCFS scheme with the results obtained from the schemes AS, QoSEF, QoSEFe, and QoS4NIP.

The E2E Actions Cost and E2E Resource Usage. Fig. 8a shows the obtained Pareto Front. The associated cost in the FCFS scheme is 0 because no TCF is deployed, and no action is currently performed on the considered NIP set-up. In the AS scheme, the cost ranges from 0.4 to 1.0 (0.48 to 1.2 in USD), and the resource usage remains 0 since no TCF is currently deployed on the considered NIP set-up. The QoSEF scheme does not induce any cost, and resource usage ranges from 0.97 to 0.99. In the QoSEFe scheme, using ANFs, resource usage has been reduced to range 0.72 to 0.90. In the QoS4NIP scheme, by combining the AS scheme and the QoSEFe scheme, the resource usage is between 0.08 and 0.63, and the cost is between 0.14 and 0.76 (i.e., 0.168 and 0.912 in USD).

The Cost-based solution selection, discussed in Section 5.5, is applied for the FCFS, AS, QoSEF, QoSEFe and QoS4NIP schemes. In general, the $(Cost_{E2E}, RU_{E2E})$ are, respectively $(0, 0)$, $(0.4, 0)$, $(0, 0.97)$, $(0.72, 0)$, and $(0.14, 0.53)$. The cost of the AS scheme is about three times higher than QoS4NIP (i.e., we have 65% financial cost-saving). Fig. 8b shows the selected E2E reconfiguration plan of each scheme. The FCFS scheme is to do nothing. The AS scheme performs Scale-up on every node (IoT Server, Cloud "GW C", Edge "GW C1", and End "GW C11"). The QoSEF scheme deploys the following VNFs: on the IoT Server a Classifier, a Shaper, and a Scheduler; on Cloud "GW C" a Classifier and a Scheduler; on the Edge "GW C1" a Classifier and a Scheduler; and on the End "GW C11" a Classifier, a Scheduler, and a Dropper. The QoSEFe scheme deploys the following ANFs: on the End "GW C11", a Classifier, a Shaper, a Scheduler, and a Dropper; and, on all other nodes (IoT Server, Cloud "GW C," Edge "GW C1"), two VNFs: a Classifier, and a Scheduler. The QoS4NIP scheme performs Scale-up on the IoT Server. It deploys, on the End "GW C11", the following ANFs: a Classifier, a Shaper, a Scheduler, and a Dropper; and, on all other nodes (Cloud "GW C," Edge "GW C1"), two VNFs: a Classifier, and a Scheduler.

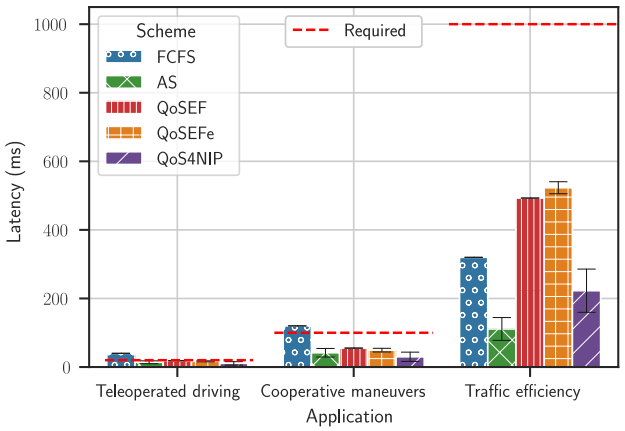
QoS provided by the optimized reconfiguration plans. Fig. 9a shows the provided E2E Latencies by the optimized reconfiguration plan of QoS4NIP versus other schemes. In the FCFS scheme, the E2E Latency for



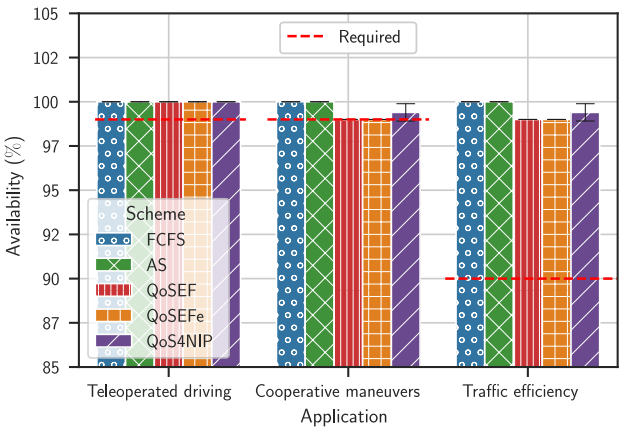
(a) Relative E2E Reconfiguration Cost and Resource Usage.

(b) Selected E2E Reconfiguration Plan (X_θ).

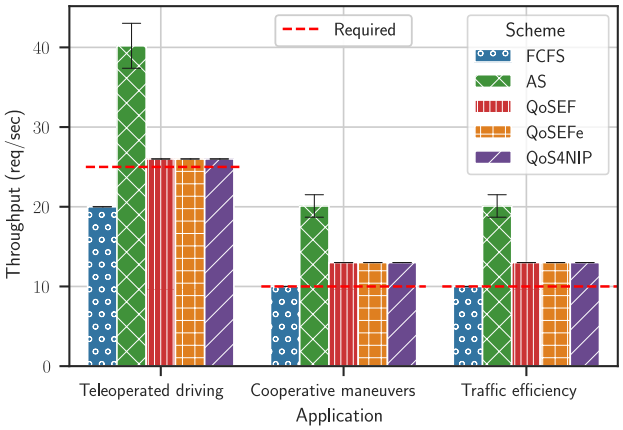
Figure 8: QoS4NIP compared with the others schemes.



(a) E2E Latencies.



(b) E2E Availability.



(c) E2E Throughputs.

Figure 9: QoS provided by the optimized reconfiguration plan of QoS4NIP versus others schemes.

Teleoperated driving is 40 ms and 120 ms for *Cooperative maneuvers*, which does not meet their requirements, 20 ms and 100 ms, respectively. Only in this scheme, the *Traffic efficiency's* required E2E Latency is reached (320 ms $\leq$ 1000 ms). In the other schemes (AS, QoSEF, QoSEFe, QoS4NIP), the E2E Latencies required by the IoT applications are sustained. However, we observe that the AS scheme provides much more than what is required by the IoT applications. For instance, for *Cooperative maneuvers*, the AS scheme provided 160 ms E2E Latency, which is more than five times what is supported by the IoT application, and that is where we see that the QoSEF, QoSEFe, and QoS4NIP schemes do better. Only by differentiating the processing of the traffic between the IoT applications, the schemes QoSEF and QoSEFe make it possible to answer the required E2E Latencies of all the IoT applications. The result is an increase in the E2E Latency of the *Traffic efficiency* ($\approx$ 500 ms), which always remains under the tolerable E2E Latency limit (under 1000 ms). The proposed QoS4NIP scheme provided the best E2E Latencies, except for *Traffic efficiency*, where the AS scheme provided low E2E Latency (160 ms).

Fig. 9b plotted the proposed E2E Availability by the optimized reconfiguration plan of QoS4NIP versus other schemes. In the FCFS and AS schemes, the E2E Availability is 100%. The fact that these schemes do not deploy droppers explains this value. However, in schemes QoSEF, QoSEFe, and QoS4NIP, the E2E Availability is 99% for *Cooperative maneuvers* and *Traffic efficiency*, due to the use of a dropper (rejecting 1% of the targeted traffic). *Teleoperated driving* being of the highest priority, its traffic is not dismissed. The E2E Availability provided by all schemes always remains under the tolerable threshold.

Fig. 9c shows the provided E2E Throughput of QoS4NIP versus other schemes. E2E Throughput required by *Teleoperated driving* is not met in the FCFS scheme. In the AS scheme, the provided E2E Throughput is much higher than the IoT application's requirement, which is not a cost-optimal plan. The schemes based on differentiation (QoSEF, QoSEFe, and QoS4NIP) use schedulers and provide the closest E2E Throughput regarding the IoT application's requirements. For instance, the QoS4NIP scheme provided to the *Teleoperated driving* an E2E Throughput of 26 req/s that is very close to the required 25 req/s.

We can conclude from these simulations that the available resources can limit the QoSEF and the QoSEFe

scheme's effectiveness in the NIP set-up. The AS scheme is effective but has not optimal costs. The QoS4NIP seems to be the best way to enable QoS for NIPs by taking advantage of the service differentiation and the autoscaling combination to overcome the above limitations of both schemes separately considered.

7 Limitations

Our proposal has some limitations. We make the following considerations about the problem at hand. First, the NIP's nodes in the Cloud/Edge are VMs and can be easily scaled (up and out). The NIP's nodes at the network End (End Gateways) are mainly hardware nodes. In rare cases, an End Gateway can be a VM located in a data center. Since the VMs are in the Cloud/Edge, the physical server's available capacity is supposed unlimited, as considered in the literature. For example, some IaaS providers are now proposing Cloud/Edge joint offers, where the limited capacity in the Edge data center is mitigated through continuous offloading to Cloud data centers. Oppositely, we consider that all reconfiguration plans generated by QoS4NIP must respect the limitation of resource capacities inside the NIP's nodes. Second, we consider only the NIP-level QoS regardless of the underlying IP-network performance (consisting of routers and switches). Thereby, the system model does not consider the network-level latency. Third, the scaling decisions are considered binary since QoS4NIP aims to minimize the scaling cost. "Zero," meaning no scaling action is necessary, and "One" meaning a scaling action is unavoidable. This allows us to be accurate in our comparison by considering the "lowest boundary" of the autoscaling approach with a minimal cost of "one new instance" at once (the non-compressible cost). Finally, we assume that only one instance of any TCF can simultaneously run on a NIP's node, and when a scale-out is applied to a NIP's node, the associated TCF will be deployed both on the initial instance and on the new replicate. We did not consider the application of different TCFs during the scale-out for the following reasons. First, we aim to maintain consistency in handling IoT traffic. When a node is scaled-out, a load-balancer is deployed upstream of the node's instances, and upon the arrival of a request, this load-balancer redirects this request to any of the node instances with no distinction. Applying different TFCs to instances would lead to an inconsistency problem for the IoT traffic handled by that node. That would result in different processing rules for requests arriving at the same (scaled-out) node. Considering such a direction will break the standard management rules for resource scaling. Indeed, we assume using the standard management rules for the scaling of the nodes, executing different TFCs in instances would be technically not sound: the scaling manager can delete any instances regardless of the TCFs executed. For these reasons, we consider, in our contribution, that any instance of a scaled-out node will process the arriving requests as decided by QoS4NIP regardless of the number of running instances. Considering the same TCFs in all instances of a given node allows us to be in line with the standard scaling approaches that proceed by deploying identical instances when scaling-out a given node and by removing any instance when scaling-in.

8 Conclusion

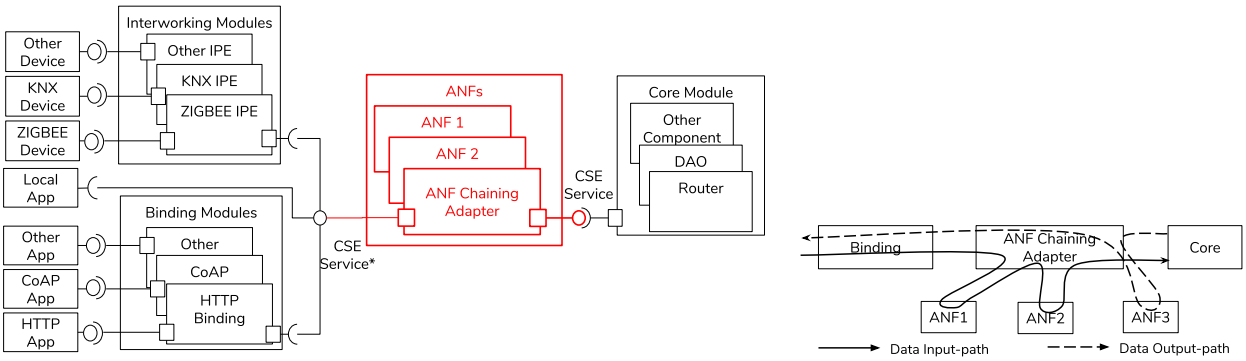
To summarise, we have proposed in this paper, a new cost-effective approach combining the advantages of the Traffic Control Functions (TCFs) deployed as NFs and the autoscaling of the virtualized processing resources. We considered the specific and challenging case of the NFV-enabled IoT Platforms (NIPs), where de facto heterogeneity is stressed by the emerging context of the recent networking technologies for routing and connectivity, the computation infrastructure for processing and storage, and the varying constraints of data producers and consumers' devices. We considered the case of the horizontal NIPs that increase the heterogeneity by addressing the cross-domain interoperability. We implemented our approach on top of OM2M, the reference implementation of the international standard oneM2M [1]. We showed by emulating different scenarios of the domain of Connected Vehicles that the classical systematic scaling can be avoided while fitting the required End-to-End QoS requirements for both common and potentially critical IoT applications. We considered the different QoS parameters (Latency, Throughput, and Availability) and the Cloud resource usage cost that we handled in a multi-objective optimization approach. We implemented TCFs that we deployed as Network Functions (NFs), which are appropriate to the capacity limits of the NIPs' nodes. We implemented a scheme, QoS4NIP, that combines the scaling actions and traffic management efficiently.

Several potential directions may be worth investigating in the future. The current QoS4NIP planner considers only the optimization of NFs (VNF/ANF) chaining to be deployed and scaling actions. It does not go further into finding the optimal parameter configuration for all these actions (scaling the NF with different sizes, adapting the loss rate within the Shaper, adapting the timeout limit, the queue reservation rate, and the other parameters for the other functions). Our algorithm needs to be extended to consider the dynamicity (i.e., the car mobility pattern in the Connected Vehicles case study) of the NIPs' nodes. This study would require additional capabilities to be taken into consideration to predict the evolution of the platform state and anticipate the related QoS violations. Moreover, in all cases, we will need an extension of the current work to include the detection of QoS violations when they are not predicted on time. The limited number of ANF-hosts prevents

from a large scale measurement campaign of the proposal's experiments. A real-world deployment in a broader scale environment would need the deployment of a large number of ANF-hosts. Today, such resources are not yet available, unlike the NFV-I that can be deployed at a significant scale by provisioning a high number of VMs (e.g., Amazon EC2 VMs). A potential future work to solve this issue is the deployment of an open crowd-sourced testbed for large-scale experimentation.

VNFs Implementation Details in Docker. We develop a prototype of the traffic functions in Java 8. Two service layer protocols are supported in this implementation: Constrained Application Protocol (CoAP) and Hypertext Transfer Protocol (HTTP). Moreover, we based the service layer protocols implementation on public optimized Open-Source libraries: Californium (https://www.eclipse.org/californium) for CoAP and Apache HTTP (https://httpd.apache.org) for HTTP. After the compilation of source code, the binaries of the TCFs are built into Docker images (Ubuntu 16.04). The associated VNF packages are created and onboarded in the ETSI-MANO OpenBaton and ready to be deployed as VNFs. IP traffic redirection, when necessary, is done using Software-defined networking (SDN) by adding Openflow rules on the NFV-I interconnection switches via the NFV-I SDN controller REST API.

ANFs Implementation Details in Eclipse OM2M. OM2M nodes are developed following a modular architectural style based on the OSGi standard [32]. Thanks to this implementation, it is possible to integrate our ANFs as OSGi Bundles. Our integration approach is achieved so that the OM2M node maintains its modular design and can operate without these new ANFs. An OM2M node (in-cse or mn-cse) is composed of the following components: *Core*, *Binding*, *Persistence*, and *Interworking Proxy Entity (IPE)*. The *Core* component is responsible for the processing of generic requests and responses (i.e., protocol-agnostic messages). It implements features such as Registration, Discovery, Re-routing, Notifications. The *Binding* components act as translators of protocol-specific messages to generic messages and vice versa. A *Binding* component is necessary for every supported protocol (i.e., HTTP, CoAP). The *Persistence* components are responsible for implementing the data storage strategy. There is an interface component, and as many as supported storage locations (in-memory, file, or server databases). Similar to the *Binding* components, they provide a translation of generic messages into non-IP (i.e., Bluetooth, ZigBee, Z-Wave) messages and vice versa.



(a) Internal structure of an OM2M node integrating ANFs.

(b) ANFs chaining in an OM2M node.

Figure 10: Seamless integration in the OM2M IoT platform.

To achieve this integration, we had to consider two options: (1) to re-implement the *Binding* components and *Interworking Proxy Entity* components of a node to add a new interface to be used for the communication with ANFs. Such a modification would have resulted in a new version of Eclipse OM2M, or (2) to use the OSGi feature "Proxying Service" [32], which allows us to intermediate an OSGi service. We have chosen the second option, which enables the integration of ANFs without affecting the oneM2M [1] standard being implemented through Eclipse OM2M. Furthermore, this option has the advantage not to change any element of the current implementation of OM2M. As shown in Fig. 10a, the main element of this architecture is the "ANF Chaining adapter." This component is specified following a design pattern [48]. It intermediates the OSGi service between the *Core* component and the *Binding* components. Depending on its configuration, it also decides to pass the request message through zero or several ANFs before reaching the *Core*. The same applies to the response message. We implemented a Management Agent (MA) that receives and installs ANF files (JAR). We also implemented an ANF deployment manager that deploys ANFs on a remote node. The deployment manager also configures ANFs dynamically, including the "ANFs Chaining adapter," which is a particular ANF. An example is illustrated in 10b. After implementing OSGi compatible source code, we generate the JAR (Java ARchive) associated with each TCFs. The generated JARs are ready to be deployed as ANFs. More details of the architecture of implementation, integration, and deployment of the TCFs into the Eclipse OM2M can be found in [45].

References

[1] oneM2M, "Technical specificationts-0002-v2.7.1: Requirements," *oneM2M*, vol. 1, pp. 1–24, 2016.

[2] ETSI, "Network functions virtualisation (nfv); architectural framework. 2014," *Group Specification*, 2014.

[3] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2015.

[4] J. G. Herrera and J. F. Botero, "Resource allocation in nfv: A comprehensive survey," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 518–532, 2016.

[5] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 73, 2018.

[6] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, p. 7, ACM, 2012.

[7] H. Liu, "A measurement study of server utilization in public clouds," in *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pp. 435–442, IEEE, 2011.

[8] A. Brogi and S. Forti, "Qos-aware deployment of iot applications through the fog," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1185–1192, 2017.

[9] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, "Resq: Enabling slos in network function virtualization," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pp. 283–297, 2018.

[10] R. Cziva and D. P. Pezaros, "Container network functions: Bringing nfv to the network edge," *IEEE Communications Magazine*, vol. 55, pp. 24–31, 06 2017.

[11] S. Palkar, C. Lan, S. Han, *et al.*, "E2: a framework for nfv applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 121–136, ACM, 2015.

[12] R. Riggio, M. K. Marina, J. Schulz-Zander, *et al.*, "Programming abstractions for software-defined wireless networks.," *IEEE Trans. Network and Service Management*, vol. 12, no. 2, pp. 146–162, 2015.

[13] K. Yasukata, F. Huici, V. Maffione, *et al.*, "Hypernf: building a high performance, high utilization and fair nfv platform," in *Proceedings of the 2017 Symposium on Cloud Computing*, pp. 157–169, ACM, 2017.

[14] M. Gallo, S. Ghamri-Doudane, and F. Pianese, "Climbos: A modular nfv cloud backend for the internet of things," in *New Technologies, Mobility and Security (NTMS), 2018 9th IFIP International Conference on*, pp. 1–5, IEEE, 2018.

[15] A. Nandugudi, M. Gallo, D. Perino, *et al.*, "Network function virtualization: through the looking-glass," *Annals of Telecommunications*, vol. 71, no. 11-12, pp. 573–581, 2016.

[16] Y. Ren, T. Phung-Duc, Y.-K. Liu, J.-C. Chen, and Y.-H. Lin, "Asa: Adaptive vnf scaling algorithm for 5g mobile networks," in *2018 IEEE 7th international conference on cloud networking (CloudNet)*, pp. 1–4, IEEE, 2018.

[17] S. Rahman, T. Ahmed, M. Huynh, M. Tornatore, and B. Mukherjee, "Auto-Scaling VNFs Using Machine Learning to Improve QoS and Reduce Cost," in *IEEE International Conference on Communications*, vol. 2018-May, pp. 1–6, IEEE, may 2018.

[18] P. Tang, F. Li, W. Zhou, W. Hu, and L. Yang, "Efficient auto-scaling approach in the telco cloud using self-learning algorithm," in *2015 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6, IEEE, 2015.

[19] S. Rahman, T. Ahmed, M. Huynh, M. Tornatore, and B. Mukherjee, "Auto-scaling network service chains using machine learning and negotiation game," *IEEE Transactions on Network and Service Management*, 2020.

[20] S. Draxler, H. Karl, and Z. A. Mann, "JASPER: Joint Optimization of Scaling, Placement, and Routing of Virtual Network Services," *IEEE Transactions on Network and Service Management*, vol. 15, pp. 946–960, sep 2018.

[21] A. N. Toosi, J. Son, Q. Chi, and R. Buyya, "Elasticsfc: Auto-scaling techniques for elastic service function chaining in network functions virtualization-based clouds," *Journal of Systems and Software*, 2019.

[22] J. Liu, W. Lu, F. Zhou, P. Lu, and Z. Zhu, "On Dynamic service function chain deployment and readjustment," *IEEE Transactions on Network and Service Management*, vol. 14, pp. 543–553, sep 2017.

[23] P. T. A. Quang, K. D. Singh, A. Bradai, and A. Benslimane, "QAAV: Quality of Service-Aware Adaptive Allocation of Virtual Network Functions in Wireless Network," in *IEEE International Conference on Communications*, vol. 2018-May, pp. 1–6, 2018.

[24] B. Yu, W. Zheng, X. Wen, Z. Lu, L. Wang, and L. Ma, "Dynamic Resource Orchestration of Service Function Chaining in Network Function Virtualizations," in *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, vol. 211, pp. 132–145, 2018.

[25] X. Cheng, Y. Wu, G. Min, and A. Y. Zomaya, "Network Function Virtualization in Dynamic Networks: A Stochastic Perspective," *IEEE Journal on Selected Areas in Communications*, pp. 1–1, 2018.

[26] V. Petrov, M. A. Lema, M. Gapeyenko, K. Antonakoglou, D. Moltchanov, F. Sardis, A. Samuylov, S. Andreev, Y. Koucheryavy, and M. Dohler, "Achieving End-to-End Reliability of Mission-Critical Traffic in Softwarized 5G Networks," *IEEE Journal on Selected Areas in Communications*, vol. 36, pp. 485–501, mar 2018.

[27] B. E. Carpenter and K. Nichols, "Differentiated services in the internet," *Proceedings of the IEEE*, vol. 90, no. 9, pp. 1479–1494, 2002.

[28] J. Ren, Y. Qi, Y. Dai, Y. Xuan, and Y. Shi, "Nosv: A lightweight nested-virtualization vmm for hosting high performance computing on cloud," *Journal of Systems and Software*, vol. 124, pp. 137 – 152, 2017.

[29] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[30] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner, "Router plugins: a software architecture for next-generation routers," *IEEE/ACM transactions on Networking*, vol. 8, no. 1, pp. 2–15, 2000.

[31] A. Panda, *A New Approach to Network Function Virtualization*. PhD thesis, EECS Department, University of California, Berkeley, 2017.

[32] O. Alliance, "Osgi service platform, enterprise specification, release 7, version 1.0," *OSGi Specification*, 2018.

[33] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[34] K. Deb, *Multi-objective optimization using evolutionary algorithms*, vol. 16. John Wiley & Sons, 2001.

[35] F. Metzger, T. Hobfeld, A. Bauer, S. Kounev, and P. E. Heegaard, "Modeling of Aggregated IoT Traffic and Its Application to an IoT Cloud," *Proc. IEEE*, vol. 107, no. 4, pp. 679–694, 2019.

[36] G. A. Carella and T. Magedanz, "Open baton: a framework for virtual network function management and orchestration for emerging software-based 5g networks," *Newsletter*, vol. 2016, 2015.

[37] M. B. Alaya, Y. Banouar, T. Monteil, C. Chassot, and K. Drira, "Om2m: Extensible etsi-compliant m2m service platform with self-configuration capability," *Procedia Computer Science*, vol. 32, pp. 1079–1086, 2014.

[38] J. Kephart, J. Kephart, D. Chess, C. Boutilier, R. Das, J. O. Kephart, and W. E. Walsh, "An architectural blueprint for autonomic computing," *IBM White paper*, pp. 2–10, 2003.

[39] D. Stiliadis and A. Varma, "Latency-rate servers: a general model for analysis of traffic scheduling algorithms," *IEEE/ACM Transactions on networking*, vol. 6, no. 5, pp. 611–624, 1998.

[40] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based non-dominated sorting approach, part i: Solving problems with box constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, pp. 577–601, Aug 2014.

[41] E. Zitzler, M. Laumanns, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm," *TIK-report*, vol. 103, 2001.

[42] H. Ishibuchi, R. Imada, Y. Setoguchi, and Y. Nojima, "Performance comparison of nsga-ii and nsga-iii on various many-objective test problems," in *2016 IEEE Congress on Evolutionary Computation (CEC)*, pp. 3045–3052, July 2016.

[43] M. Boban, A. Kousaridas, K. Manolakis, J. Eichinger, and W. Xu, "Connected Roads of the Future: Use Cases, Requirements, and Design Considerations for Vehicle-to-Everything Communications," *IEEE Veh. Technol. Mag.*, vol. 13, no. 3, pp. 110–123, 2018.

[44] D. Hadka, "Platypus: A free and open source python library for multiobjective optimization," *Available on Github, vol. https://github. com/Project-Platypus/Platypus*, 2017.

[45] C. A. Ouedraogo, S. Medjiah, and C. Chassot, "A Modular Framework for Dynamic QoS Management at the Middleware Level of the IoT: Application to a oneM2M Compliant IoT Platform," in *IEEE Int. Conf. Commun.*, vol. 2018-May, pp. 1–7, IEEE, may 2018.

[46] C. A. Ouedraogo, E. Bonfoh, S. Medjiah, C. Chassot, and S. Yangui, "A prototype for dynamic provisioning of qos-oriented virtualized network functions in the internet of things," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pp. 323–325, June 2018.

[47] K. Hwang, X. Bai, Y. Shi, M. Li, W.-G. Chen, and Y. Wu, "Cloud performance modeling with benchmark evaluation of elastic scaling strategies," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 130–143, 2015.

[48] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.