# A Model for Quantifying Performance Degradation in Virtual Network Function Service Chains

Ricardo J. Pfitscher, Arthur S. Jacobs, Eder J. Scheid, Muriel F. Franco, Ricardo L. dos Santos,
Alberto E. Schaeffer-Filho, Lisandro Z. Granville
Institute of Informatics – Federal University of Rio Grande do Sul
Av. Bento Gonçalves, 9500 – Porto Alegre, Brazil
Email: {rjpfitscher, asjacobs, ejscheid, mffranco, rlsantos, alberto, granville}@inf.ufrgs.br

***Abstract*—Virtual Network Functions (VNFs) can be chained and provisioned on demand, providing elasticity and dynamicity to the network. Due to the interdependencies between VNFs, resulting service chains may not work as expected, and because of that, it is crucial to determine which VNFs are having a negative impact on the service quality. In this paper, we introduce a model to quantify the `guiltiness` of a VNF on being a bottleneck in a service chain, which provides a metric that estimates the impact on processing delay. In addition, we propose an adaptive algorithm, based on linear regression and neural networks, to adjust the model parameters according to the environment particularities, such as the type and number of VNFs. We show through an experimental evaluation that the `guiltiness` metric faithfully characterizes end-service performance, by identifying up to 94% of the bottleneck VNFs in the analyzed scenarios. Also, we provide artifacts for researchers to reproduce our results in other scenarios.**

## I. Introduction

Network Functions Virtualization (NFV) allows the virtualization and deployment of virtual network middleboxes in Commercial-Of-The-Shelf (COTS) servers by decoupling network functions from the underlying hardware [1]–[3]. In NFV, Virtual Network Functions (VNFs) are chained and provisioned on demand, providing elasticity and dynamicity to the network. In particular, NFV Service Providers (SPs) use forwarding graphs to specify service chains and to provide customized network services (NSes) to meet individual client demands [4]. For instance, an SP can provide a security service by chaining security related VNFs (*e.g.*, firewall, deep packet inspection, and intrusion detection system), or chain performance-related VNFs (*e.g.*, load balancer, WAN optimizer, and video cache) to offer services that can improve network performance.

Because of the dependency among VNFs that form a service chain, any VNF can have a negative impact on the network service performance [5]. As a consequence, network operators should be able to identify the operational state of VNFs that are part of the chains, and check whether the service chains are performing as expected (*i.e.*, meeting predefined requirements). If not, one must apply a bottleneck identification method to pinpoint which VNFs are degrading the performance of the chain.

978-1-5386-3416-5/18/$31.00 © 2018 IEEE

A common method for discovering bottlenecks is to monitor the physical and virtual resources in the chain [6]. However, NFV environments have important particularities: (*i*) the diversity of network function types [1], with distinct metrics to be evaluated, prevents the use of traditional bottleneck diagnostic methods [7]; (*ii*) in the Network as a Service (NaaS) model [8], NFV SPs may not have knowledge of which end-services are being accessed by the customers, due to privacy reasons. This lack of information restricts the measurement of end-services' metrics, which turns performance prediction even more challenging; and (*iii*) there are VNFs that rely on non-blocking I/O system calls to avoid context switching, resulting in a CPU utilization always close to 100%, which hinders the use of a bottleneck detection method based solely on CPU utilization [4].

In this paper, we address such particularities by proposing a model to quantify the `guiltiness` of each VNF in a given chain on being a bottleneck. To establish a reduced set of performance-representative metrics being monitored (*i*), we combine concepts from queueing theory and empirical observations to model the metric based on *CPU usage*, *active time*, and *network queuing*. To soften the need of end-services measurements (*ii*), we estimate the impact on processing delay of each VNF through a weighted sum function. To address the cases in which the performance of the VNFs cannot be accurately predicted (*iii*), we propose an algorithm to adapt the weights of the model. In particular, nonlinear regression and neural networks are combined to establish the coefficients that faithfully represent the performance of each VNF. To provide evidence of the model effectiveness, we perform an experimental evaluation with synthetic bottlenecks. Results show that `guiltiness` accurately represents the end-service performance, detecting 94% of the bottlenecks. Besides, the model achieves a 0.98 R-squared when predicting the response time.

The remaining of this paper is organized as follows. In Section II, we discuss related work. In Section III, we model `guiltiness`. In Section IV, we discuss the adaptive algorithm. We describe an architecture to learn coefficients and monitor the `guiltiness` of each VNF in Section V. Experimental evaluation and results are described in Section VI, as well as directions for reproduction artifacts. Finally, we conclude and provide future directions in Section VII.

## II. Related Work

Bottleneck detection using queueing network models has been well studied in the literature in distinct contexts (*e.g.,* Web servers [9], virtualized applications [10], and production networks [11]). However, in the case of NFV service chains, the research is still in its infancy and the studies that have been carried out to detect bottlenecks [4] [12] [13] present major limitations, as follows.

Wu *et al.* [4] designed PerfSight, a system to diagnose three types of NFV performance issues: resources misallocation (bottlenecks), competition for shared resources (contention), and design implementation (performance bugs). The authors consider the software data plane as a pipeline of elements that read, process, and write traffic to another element's buffer, similarly to the concept of queueing networks. If an element cannot write to its successor or the target buffer is full, packets will be dropped. Thus, the system applies the *packet loss of each element* in the software data plane to locate resource contention points, and identify bottlenecks as well. Our study differs from PerfSight in two major aspects. First, PerfSight seeks to only pinpoint which resource is responsible for contention, while we wish to quantify the impact that each VNF has on the performance of the service chain, and then find the bottlenecks. Second, their approach uses packet drops as the main metric, while we consider that queueing occurs before packet drops. Therefore, we measure the average queue size in the network interface.

Gember *et al.* [12] propose an architecture for orchestrating middleboxes in cloud environments. The authors claim that there is an increase in the number of middlebox applications running in the cloud, and design a new orchestration layer with the purpose of composing chains, scaling capacities, and tuning network functions interactions. Regarding capacity scaling, the authors indicate a need for bottleneck identification methods and propose an application-aware heuristic that considers the performance reported by end-services to provision more resources or redistribute traffic. The proposed heuristic consists in an iterative process: for each middlebox in the chain, it adds instances until application performance is improved. Although this approach allows the orchestrator to sustain the desired applications' performance level, the dependency of the iterative process on the end-service measurements hinders the discovery of the root cause of problems. Even though an orchestrator can determine that the bottleneck is the middlebox that causes more performance improvement for the applications, it is difficult to quantify the effect of each middlebox, impacting the quality of planning decisions.

Sauvanaud *et al.* [13] propose a solution for anomaly detection and root cause identification. The proposal combines black-box and gray-box monitoring data with machine learning mechanisms to detect anomalous situations. The root cause of problems detection derives from a database trained with both monitoring metrics and SLA violations. A detection model compares the behavior of VNFs through a knowledge database to result in a classification probability. Next, a probability threshold defines whether a VNF is performing anomalously; if true, such VNF is probably the root cause of problems. Although this approach provides accurate results for root cause identification, it depends on correct classification of anomalous situations in the knowledge database. In our case, we present a model independent of previous classification, where we learn about the VNFs behavior to adjust estimations accordingly. Also, our model is able to quantify the impact of each VNF on the overall performance, which is not necessarily an anomalous situation.

## III. Guiltiness Modeling

In an NFV scenario, as exemplified in Fig. 1, flows from clients to end-servers pass through $n$ VNFs that form a service chain. Each VNF performs a specific function that encompasses receiving, processing, and forwarding packets to the next VNF. Each VNF has a specific impact on the overall NS performance. We propose the `guiltiness` model as a metric to quantify the impact on the processing delay.

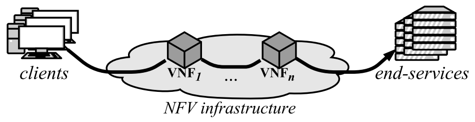


Figure 1. Network Functions Virtualization scenario

A plethora of metrics is available to analyze NFV performance in the literature. However, we argue that these metrics provide inconclusive results for bottleneck detection when analyzed in an isolated manner. One of the reasons for that is the heterogeneity of network function behaviors. To illustrate, observe the charts presented in Fig. 2. We create a simple NFV scenario where a demand of requests from a client to a Web server passes through one VNF. Fig. 2(a) depicts the requests demand, and the throughput measured when the VNF runs a firewall, or when it runs an IDS. Figs. 2(b) and 2(c) present the measurements of resources usage (*i.e.*, CPU, memory and NIC), cache miss rate, and network queuing during workload execution the IDS and the firewall, respectively. As one can notice, it is hard to relate metrics to performance. While the firewall achieves a higher throughput than the IDS, it has a higher amount of packets queued on the NIC and a lower CPU consumption. Also, while some metrics are not affected by demand (*e.g.*, cache miss), others follow the behavior in only one of the VNFs (*i.e.*, committed memory).

The `guiltiness` of a VNF is the combination of four metrics (Eq. 1): *network queue length* ($Q$), *average queue usage* ($Qu$), percentage of *CPU usage* ($U$), and *active percentage time* ($A$). Queuing can occur in two situations: when the transmission demand of VNFs is greater than the available bandwidth, or when the processor throughput cannot accommodate the rate of packets being received [4]. While queue length ($Q$) is the amount of bytes queued in the virtual NIC (vNIC), queue usage ($Qu$[1]) is the ratio between $Q$ and the

[1]Note, to avoid any confusion with the resource usage from queueing theory, we use $Qu$ instead of $U_{NIC}$.

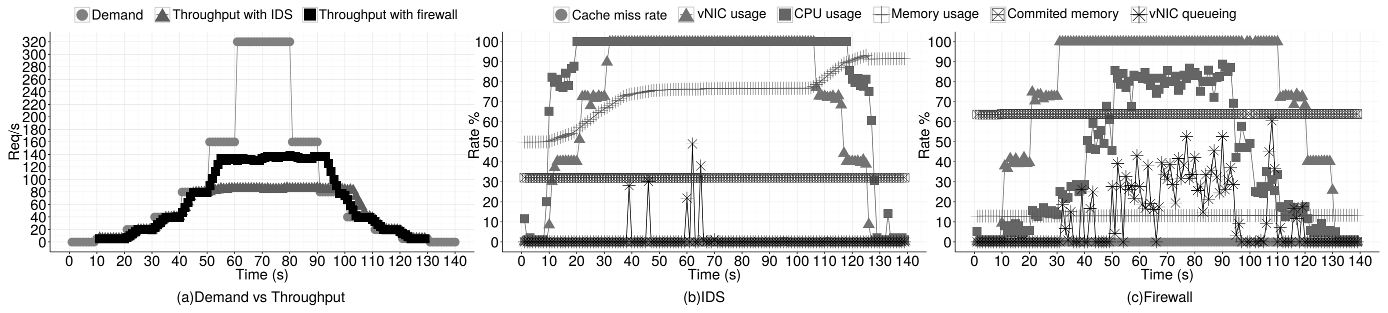


Figure 2. Throughput and metrics measurements on distinct VNFs when submitted to the same workload.

size of the vNIC's buffer. CPU usage ($U$) quantifies how busy the CPU is, as a percentage of its capacity. Active percentage time ($A$) is a metric adapted from the active duration time proposed by Roser *et al.* [14]; it represents the fraction of the time that the CPU computes something relevant.

Taking these metrics into account, we define the `guiltiness` ($G$) of a VNF $v$ on the performance degradation of a service chain as:

$$G_v = w_1 \frac{1}{1-U_v} + w_2\, A_v + w_3\, Qu_v - w_4 \frac{A_v}{1+Q_v} \tag{1}$$

where, $w_1$, $w_2$, $w_3$, and $w_4$ are the weights of each term. These weights define the relevance of each factor on `guiltiness`. We discuss how to define these values in Sec. IV. Table I describes the symbols used in the remainder of this paper.

Table I
KEY TERMS AND DESCRIPTIONS

| Term | Description | Domain | Interval |
|---|---|---|---|
| $A$ | active percentage time | $\mathbb{R}$ | [0,1] |
| $c$ | a given service chain | - | - |
| $C$ | a counter for $U$ values | $\mathbb{Z}$ | [0,$P$] |
| $G_v$ | `guiltiness` of VNF $v$ | $\mathbb{R}$ | [0,1] |
| $G_c$ | `guiltiness` of chain $c$ | $\mathbb{R}$ | [0,1] |
| $\lambda$ | arrival rate | $\mathbb{R}$ | [0,$\infty$) |
| $P$ | monitoring window size | $\mathbb{Z}$ | [0,$\infty$) |
| $Q$ | network queue length | $\mathbb{Z}$ | [0,$\infty$) |
| $Qu$ | network queue usage | $\mathbb{R}$ | [0,1] |
| $R$ | residence time | $\mathbb{R}$ | [0,$\infty$) |
| $Rt$ | response time | $\mathbb{R}$ | [0,$\infty$) |
| $RtNorm$ | normalized response time | $\mathbb{R}$ | [0,1] |
| $T$ | threshold for active CPU usage | $\mathbb{R}$ | [0,1] |
| $S$ | service time | $\mathbb{R}$ | [0,$\infty$) |
| $U$ | CPU usage | $\mathbb{R}$ | [0,1] |
| $v$ | a given VNF | - | - |
| $X$ | throughput | $\mathbb{R}$ | [0,$\infty$) |
| $w_i$ | weight coefficient $i$ | $\mathbb{R}$ | [0,$\infty$) |

Because our goal is to estimate the impact on the processing delay that a given VNF $v$ imposes on the service chain $c$, instead of using conventional queueing theory models [15]–[17], we combine the metrics in a weighted sum, providing a normalized delay estimation. We abstract queueing models and use adjustable weights in each metric to account for individual characteristics. In fact, our model is based on the well-accepted Utilization Law [17], extending it with other factors based on both state-of-the-art findings and empirical observations through experimental analysis. In the remainder of this section, we explain how each metric contributes to the `guiltiness` estimative model.

## A. CPU Utilization

The Utilization Law defines the residence time $R$ of a request in a node as the relation between service time $S$ and CPU usage $U$ (Eq. 2). Utilization impacts $R$ in an exponential manner. This means that at low values, $U$ has a small impact on the performance, but as $U$ increases the performance falls quickly. This happens because, at high utilization, the CPU cannot cope with the processing demand, thereby leading to the queuing of processing tasks. To avoid the need of measuring $S$ values of heterogeneous services, we define an approximation and set the coefficient $w_1$ as the term's importance in `guiltiness`. This approach allows us to adjust the coefficient according to different network functions behavior.

$$R_v = \frac{S_v}{1-U_v} \cong w_1 \frac{1}{1-U_v} \tag{2}$$

## B. Active percentage time

The second term relates to the active percentage time. We rely on the study of Wang *et al.* [11], who discuss the active duration time as a metric to detect bottlenecks. The authors show that this metric faithfully characterizes bottlenecks in production networks. The original meaning of active duration time corresponds to the period of time that a node is in a non-idle state. However, as the CPU can show fluctuations of low consumption, we use a threshold to establish when the CPU is idle. $A$ is measured in the following way: a monitor collects $P$ samples of CPU usage and for each value greater than a threshold $T$ it increments a counter $C$. Afterwards, $A$ is computed as the ratio between $C$ and $P$. The threshold $T$ is set by network operators according to their perspective of relevant load. In our case, through empirical observations, we define $T$ as 0.2. The following example helps to understand how $A$ is computed. Consider two time-series with measurements of $U$ from distinct VNFs: $U_{vnf_1}$:$[0.3, 0.4, 0.2, 0.3]$ and $U_{vnf_2}$:$[0.1, 0.5, 0.1, 0.5]$. Both VNFs have an average usage of

0.3, which means that they are not overloaded. However, while $A_{vnf_1}$ is equal to 1.0 ($C_{vnf_1} = 4, P = 4 \rightarrow A_{vnf_1} = 4/4 = 1.0$), $A_{vnf_2}$ is 0.5 ($C_{vnf_2} = 2, P = 4 \rightarrow A_{vnf_2} = 2/4 = 0.5$). Thus, $vnf_1$ has a higher impact on the network service performance because it is twice as active as $vnf_2$. Thus, we define that `guiltiness` is proportional to $A$ and $w_2$ determines its impact.

## *C. Network interface card queueing*

The occurrence of network queuing indicates that a VNF is not being able to cope with the incoming demand. Thus, we add the average queue usage ($Qu$) to the model. Eqs. 3 and 4 compute the queuing impact on `guiltiness`. According to the Utilization Law, the current queue length ($Q$) depends on $U$. If we isolate $U$ in Eq. 2 and replace it in Eq. 3, we obtain the residence time as a function of the queue length times the service time. Thus, if we limit the queue size to the vNIC's buffer size, we can correlate the queue utilization ($Qu$) to the residence time (Eq. 4).

$$Q = \frac{U}{1-U} = \frac{1 - S/R}{1 - (1 - S/R)} = \frac{R}{S} - 1 \rightarrow R = (Q+1)\,S \quad (3)$$

$$\begin{aligned} Qu_v &= Q_v / buffer_v \\ R_v &= w_3\, Qu_v \end{aligned} \quad (4)$$

Although we rely on the same arguments as Wu *et al.* [4] for using measurements of vNIC queueing as effective signals of performance degradation, one can argue that CPU usage observation is enough for bottleneck identification because when network queuing occurs, it implies in high CPU usage. To defend such argument, let us delve on queuing modeling: instead of using our abstracted single queue model, consider that a VNF is modeled as two processing queues, one NIC, and one CPU, as depicted in Fig. 3. The NIC stores requests in a buffer, processes these requests according to the NIC's nominal capacity, and then forwards these requests to the CPU queue.

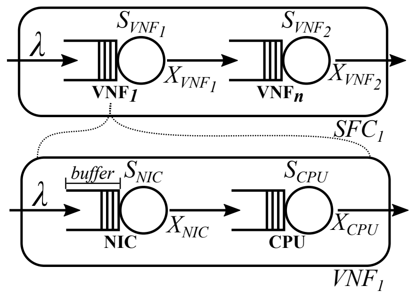


Figure 3. Network queuing VNF modeling

Now, suppose a VNF where CPU service time $S_{CPU}$ is at least two times smaller than the NIC service time $S_{NIC}$. In such case, a 100 Mbps NIC will process a 10 KB request in 0.8ms ($\frac{10*10^3}{12.5*10^6}$) and the CPU will take 0.4ms to process the same 10 KB request. When this VNF is submitted to an arrival rate $\lambda$ greater than 1250 req/s, NIC usage will be 100% (Utilization Law, $U = S \times \lambda$), and requests will be queued in the buffer. However, as the CPU will receive requests at the rate of the NIC throughput $X_{NIC}$, which in the best case is 1250 req/s, the CPU usage will be 50%. This example shows that even when $U_{CPU}$ is not high, queueing can occur.

## *D. Active time for queued requests*

The last term in $G_v$ corresponds to the active time taken by a node to process queued packets: the shorter the active time spent with queued packets, the lower the `guiltiness` of that VNF. Hence, we subtract a value proportional to the active time and the queue length. The composition of $w_2$ and $w_4$ results in the following outcome: the longer the queue length, the higher the impact of active time on `guiltiness`. Traditional theoretical models assume that queue size increases as CPU usage increases; however, this may not be the case. Instead, we advocate that to measure `guiltiness` accurately, it is necessary to combine both CPU and queue observations into a single metric. Thus, by leveraging the third and fourth terms we can adjust the value for `guiltiness` according to the queue measurements.

As an example, consider the following situation: two VNFs, both having high active percentage time ($A = 1.0$), low average CPU usage ($U = 0.3$), and only one of them presenting network queue usage ($Qu = 0.2$). If we neglect the fourth term, and consider the same value (*e.g.*, 0.2) for the weights $w_1$, $w_2$, and $w_3$, the `guiltiness` value is 0.52 (0.28 + 0.2 + 0.04) for the case with network queuing and 0.48 (0.28 + 0.2) for the case without queuing. This result shows that both VNFs have similar impact on the end-service performance. However, when queuing occurs in the NIC buffer, the performance degrades severally. As we previously explained, queuing in the buffer occurs when the interface card is not able to cope with the incoming demand. If we consider that a 100 Mbps NIC has a service time of 0.8ms for a 10 KB request, when its usage is close to 100% the residence time is about one second ($\frac{0.0008}{1-0.999}$). Nevertheless, when there is no queuing, this residence time is close to 4ms for an 80% usage of the NIC processing capacity (no queuing on the buffer). This difference is too large to allow the comparison between situations where network queuing is present and where it is not.

To work around the equivalence between metrics, and limit the impact of $A$ on `guiltiness` according to network queuing ($Q$) occurrence, we subtract the rate between $A$ and $Q$. Assume that the performance impact of $A$ must be greater when the CPU activity is related to the processing of network queued requests. This means that, as the number of queued requests on the NIC buffer ($Q$) grows, the relative impact of $A$ on `guiltiness` must grow proportionally. The opposite is also true: the smaller the $Q$, the smaller the impact of $A$ on performance. Hence, in the fourth term of the `guiltiness` metric, we restrict the impact that $A$ has on our performance degradation metric. This impact is limited to

$Q$: by Eq. 4, $Q = Qu \times buffer$, and as $buffer \to \infty$, then $\lim_{Q\to\infty}(\frac{A}{Q+1}) = 0$ and $\lim_{Q\to 0}(\frac{A}{Q+1}) = A$.

Consider the same case from the previous example: $A_{vnf_1} = A_{vnf_2} = 1.0$, $U_{vnf_1} = U_{vnf_2} = 0.3$, $Qu_{vnf_1} = 0.2$, $Qu_{vnf_2} = 0$, and $w_1 = w_2 = w_3 = 0.2$, which resulted in $G_{vnf_1} = 0.52$ and $G_{vnf_2} = 0.48$. Now, consider the subtraction of rate $\frac{A}{Q+1}$ as a term, with $w_4 = 0.2$. In such a case, the resulting guiltiness values for a $buffer$ size of $1.5 \times 10^6$ bytes are $0.52$ $(0.28 + 0.2 + 0.04 - 0.6 \times 10^{-6})$ and $0.28$ $(0.28 + 0.2 - 0.2)$ for the cases with and without queued requests, respectively. The difference of $0.24$ is six times greater than the previous one ($0.04$), which corroborates our claim that the rate between $A$ and $Q$ represents a limit to the impact of $A$. Note that using $w_2 = w_4$ eliminates the impact of $A$ on guiltiness when there are no network queued requests. To circumvent a contradiction with our claim for $A$, we use the constrain of $w_2 > w_4$

## IV. Guiltiness Weights

Weights in Eq. 1 define the relevance of each factor to guiltiness. These values can vary because of individual aspects of service chains and network functions. For example, if a network operator detects that CPU usage has a higher impact on service performance than the active time metric, he/she can increase $w_1$ and reduce $w_2$. Also, NFV environments are subject to frequent changes regarding traffic patterns and provisioned resources. These aspects may also require the weights to be adjusted accordingly.

Let us assume that a VNF service chain is a queueing network model. As such, we can adapt queueing network laws to our purpose. First, consider that the residence time is the time that a given VNF takes to process a request; then, according to the General Response Time Law, the sum of the residence time of each node in the chain is the total chain residence time. Therefore, if we take into account that $G_v$ is a representation of residence time in VNF $v$, the sum $G_c$ of all $G_v$ values in a service chain $c$ is the total residence time. In Eq. 5, we derive $G_c$ to extract the weights of guiltiness. Note that $w_1$, $w_2$, $w_3$, and $w_4$ are constant for all $v$ in $c$.

$$\begin{aligned} G_c =& \sum_{v\in c}^{n} G_v \\ =& w_1 \sum_{v\in c}^{n} \frac{1}{1-U_v} + w_2 \sum_{v\in c}^{n} A_v + w_3 \sum_{v\in c}^{n} Qu_v - w_4 \sum_{v\in c}^{n} \frac{A_v}{1+Q_v} \end{aligned} \tag{5}$$

where $n$ is the number of VNFs in the chain $c$.

The equation has the following outcome: to fit $G_c$ as a representation of the chain residence time $R_c$, we must find a quadruple $<w_1, w_2, w_3, w_4>$ that makes the relation exact to the metric observations. To achieve that, the first step is to aggregate the measurements of $<U, A, Qu, Q>$ and compute the sum of each term. Then, each of the four terms in Eq. 5 $<t_1 = \sum_{v\in c}^{n} \frac{1}{1-U_v}, t_2 = \sum_{v\in c}^{n} A_v, t_3 = \sum_{v\in c}^{n} Qu_v, t_4 = \sum_{v\in c}^{n} \frac{A_v}{1+Q_v}>$, as well as, the respective $<Rt_c>$, are stored in a database. Then, by analyzing historical information we can compute the coefficients that make $G_c \cong R_c$.

We rely on a hybrid learning procedure, combining both neural networks and nonlinear regression, to compute the coefficients. The reason why we combine these methods is to avoid fluctuations of regressions results. Considering that not all scenarios are available a priori, and that non-linear regressions take into account the entire dataset of measurements, outliers can directly effect weights estimation. On the other hand, neural networks perform more abstracted regressions, which make them less sensible to outliers. The learning procedure consists of two phases: *training* and *working*. Training data consists of metrics and coefficients. Training can be either manual (the operator inserts training data) or automatic (using measurements and regressions). Once there is enough information to perform the predictions, the learning procedure enters into the *working* phase. At the working phase, the hybrid procedure combines results from neural network and regressions to compute the coefficients. Algorithm 1 presents the learning procedure using automatic training.

**Algorithm 1** adaptive_guiltiness

```
1: collect($t_1, t_2, t_3, t_4, Rt$);
2: $RtNorm = Rt/RtMax$;
3: store($t_1, t_2, t_3, t_4, RtNorm$);
4: if not training_phase then
5:     coeff[nn] = learn(neural_net,$t_1, t_2, t_3, t_4, RtNorm$);
6: coeff[nlr] = learn(nonlinear_regression,metrics,$G_c$);
7: compute_Rsquared(current, default, nn, nlr);
8: if rsquared[nlr] $>$ trshold then
9:     store([$t_1, t_2, t_3, t_4, RtNorm$],coeff[nlr]);
10: if rsquared[nn] $>$ trshold then
11:     store([$t_1, t_2, t_3, t_4, RtNorm$],coeff[nn]);
12: index = $maxRsquared$ in {current, default, nn, nlr};
13: coeff[current] = coeff[index];
14: rsquared[current] = coeff[index];
15: update(coeff[current]);
```

Algorithm 1 starts by calling *collect()* to capture the metric values from each VNF. Next, it computes the normalized response time and calls *store()* to save the values into a training database. Once the *training* phase finishes, in line 5, the algorithm calls *learn()*, which uses a neural network to predict the coefficients based on the training data. Next, in line 6, the algorithm calls *learn()* again, but this time using the nonlinear least squares regression to provide coefficients that fit the guiltiness function with the metric history. Afterwards, the algorithm checks the guiltiness accuracy. To do so, it computes, in line 7, the coefficient of determination (R-squared) of the fit using each of the computed coefficients (default, current, neural network, and nonlinear regression). If the computed R-squared values are greater than a threshold (lines 8 to 11), the knowledge base of the learning procedure is updated to include a new match between the measured values and the computed coefficients ([$t_1, t_2, t_3, t_4, RtNorm$], $coefficients$). Finally, the algorithm calls *update()* to renew the coefficients used by each VNF to compute guiltiness.

Although we propose an adaptive algorithm to define the model weights, it depends on the capacity of operators to monitor end-metrics. Through empirical analysis of specific cases, it is possible to estimate default values for these coefficients. In particular, for the experiments reported in Sec. VI, we found that $w_1 = 0.01$, $w_2 = 0.9$, $w_3 = 0.001$, and $w_4 = 0.8$ provide small `guiltiness` (1%) for low values ($U = 0\%, A = 0\%, Qu = 0\%, Q = 0$) and up to 100% for critical values (*e.g.*, $U = 100\%, A = 100\%, Qu = 100\%, Q = 150000$). The lower values of $w_1$ and $w_3$ are due to the need of $G$ being limited to 1.0, and to $w_4$ already covering queueing impact, respectively.

## V. Guiltiness Monitoring

Our approach for monitoring `guiltiness` consists of two key processes: one for learning coefficients and another to maintaining the `guiltiness` status up-to-date in the VNF Managers (VNFM). Figure 4 depicts the elements and their components to implement these processes, which includes: (*i*) the *`guiltiness` Learning Framework* with two independent services, a collector and a learner; (*ii*) the *Monitoring Agent* that runs inside each VNF; and (*iii*) the ETSI NFV Management and Orchestration (MANO) reference framework.

The learning service enables the Algorithm 1 and runs as follows. A set of monitoring agents measure the model's metrics (*guiltiness_data*), which are requested by a collector service at predefined intervals (*coll_period*). The collector service aggregates the monitoring data (*aggr_data*) of each service chain, *i.e.*, the computed sums of the terms $t_1$, $t_2$, $t_3$, $t_4$ and the *end_metric* value of the *end_service*. After, the collector service sends the aggregated information about the *service_chain* to the learning service, through Rest API calls. This information includes *aggr_data*, *end_service* details, and *vnf_fg* with monitoring agent addresses. When the learning service receives the aggregated information, it first stores *aggr_data* into the service chain history (*sc_history*). It then executes the learning algorithm to determine the coefficient values. Finally, the learning service updates the coefficients into each monitoring agent.

For maintaining the `guiltiness` status up-to-date in the VNF Managers, we leverage an active monitoring approach [6] and a result-aware monitoring proposed in our previous work [18]. In summary, monitoring agents run a diagnosis model. The diagnosis consists in comparing the average of measured `guiltiness` time series (*g_data_ts*) to a threshold (*upper_thr*), and then establishing the VNF's state (*g_state*). When agents observe that the state of a VNF has changed, they send the result to the VNFM. For example, an operator can configure the *upper_thr* to 10%, which means that such a VNF may be responsible for up to 10% of the total response time. In a situation in which `guiltiness` measurements reach this value, the agent notifies the VNFM about the possibility of chain performance degradation. This approach allows managers to rapidly detect bottlenecks and perform actions to reduce performance problems (*e.g.,* provisioning more resources).

Although our proposed architecture has been designed considering active monitoring, the use of a REST API also allows managers to request VNF information from agents in distinct ways, *e.g.,* by polling the VNFs of a service chain. Thus, managers can have a view of all `guiltiness` values in the chain, compute the value of $G_c$, and pinpoint the bottleneck.

## VI. Experimental evaluation

To evaluate the ability of identifying bottlenecks, we defined a scenario with two VNFs and three network flows. Both VNFs run in isolated Virtual Machines (VMs). VNF$_1$ runs a Firewall blocking UDP traffic, and VNF$_2$ runs a Deep Packet Inspection (DPI). The flows consist of a 5 Mbps UDP traffic, generated by `iperf`; a 30 Mbps video stream background traffic, generated by `ffmpeg`; and HTTP traffic, produced by RUBiS [19]. Each iteration of the workload runs for 90 seconds. To simulate bottlenecks, we varied the processor and network capacities, relying on two Xen configuration parameters to specify resource provisioning: `cap` (%), which limits the CPU usage, and `rate` (Mbps), which limits the network output rate.

We measured the average response time of RUBiS as the end-service metric ($Rt$). For each combination, we performed ten repetitions, which resulted in a 95% confidence level for a 10% measurement error. We captured CPU usage through `mpstat`; the queued size, with the `tc` tool from Linux; and the $Qu$ value was measured as the ratio between the average queue size (bytes) and the size of the vNIC's buffer (1.5 MB in our tests). The neural network runs the `l-bfgs` algorithm, with two layers, converging from 100 to 5 neurons. The threshold for inserting weights into the database (*rsquared_thr*) was set to 0.8. The threshold for considering a VNF as active (*active_thr*) was set to 0.2. The training phase runs until it reaches twenty entries (*training_thr*). The current monitor implementation was developed in Python and is publicly available on GitHub[2]. Also, to avoid division by zero, we adjust the $w_1$ denominator to $(1.001 - U)$.

### A. Evaluation results

We first assess how each of the metrics in `guiltiness` relates to the end-service performance, allowing us to compare our results to related work. $U$ is one of the several metrics considered by [13] in their classification mechanism. Although $A$ still have not been used in the context of NFV, Wang *et al.* [11] defined it as meaningful for detect bottlenecks in production networks context. $Qu$ relates to the occurrence of packet drops used by PerfSight [4], because NIC queuing occur just after packet drops. Stratos [12] uses $R$ in an iterative process to adjust resources of VNFs, we use response time to observe whether metrics are representative of end-service performance. Each plot in Fig. 5 correlates the values for a specific metric from the two VNFs, considering the possible combinations of the allocated resources (`cap` – 10%, 50%, and 100%, and `rate` – 10 Mbps, 50 Mbps, and 100 Mbps). The measured response time varies according to the point size and color. Due to the proportional behavior of $Qu$ and $Q$ (see Eq. 4), we only plot $Qu$ values. As results show, the traditional metrics (Figs.

[2]https://github.com/ricardopfitscher/genic

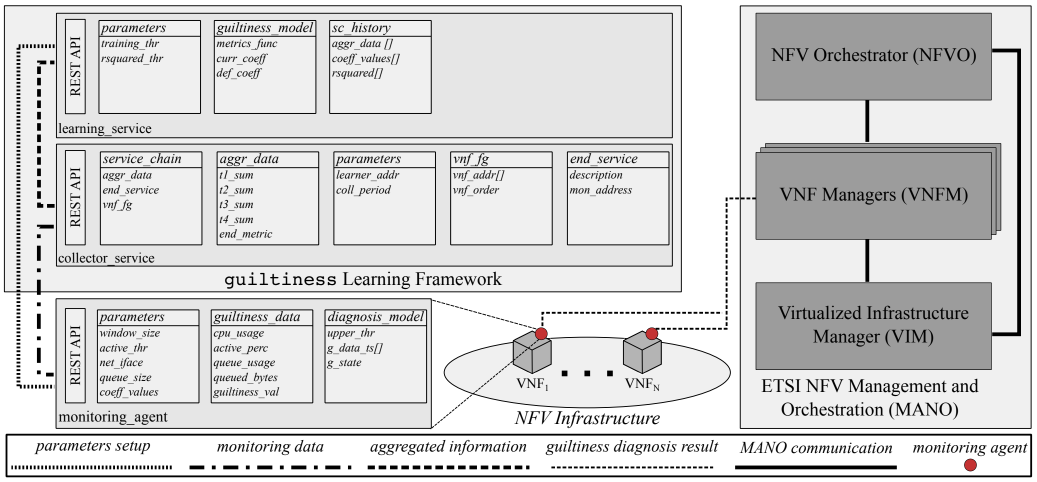


Figure 4. `guiltiness` monitoring architecture and interactions with ETSI MANO. Two services run in the *Learning Framework*, one for collecting measurements from agents and other for learning metric impacts and determine coefficients.

5(a), 5(b), 5(c)) are unable to reliably express the end-service performance, because often the measurements are ambiguous (highlighted squares): for two similar measured values, these metrics result in both low and high $Rt$ values. In turn, the `guiltiness` metric (Fig. 5(d)) faithfully characterizes the end-service performance. As such, when both VNFs have low $G$ values, the measured response time is also low. If $Rt$ increases, $G$ grows accordingly.

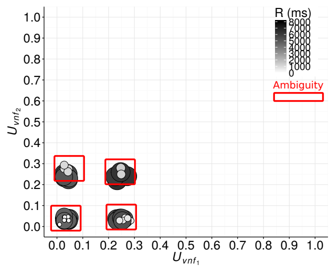
Considering CPU usage (Figure 5(a)), even with low consumption, there are four ambiguous zones in the graph. For instance, consider the case with $U_{vnf_1}$=0.05 and $U_{vnf_2}$=0.25. In such a case, the response time has both high values (close to 4000ms) and low values (close to 1000ms). The ambiguity is even worse in *quasi-zero* usage cases, that resulted in either 2000ms or 242ms. Although the active percentage time (Figure 5(b)) shows a distinct behavior from the CPU usage, the ambiguities also appear. For example, if we set $A_{vnf_1}$ to 0.25 and follow the $y-axis$, the expected behavior would be an increase in $Rt$. However, at $A_{vnf_2} = 0.85$, there are both high and low response time situations. The queue observations (Figure 5(c)) provides a more significant result: when queuing occurs, the response time is high. However, two problems arise when observing this chart: it is not possible to quantify how much each VNF impacts on the performance, and there are cases with $Qu = 0.0$ related to high response times. All these ambiguities emphasize the need of combining the metrics to have an accurate diagnosis of bottlenecks.

To get a clearer idea of how the `guiltiness` metric can be used to identify bottlenecks, consider the four samples presented in Table II. In case #1, the network capacity limits the network flow at the first VNF, and, as a consequence, queuing occurs, thus resulting in a `guiltiness` of 50% for VNF$_1$. In this case, an analysis based on CPU usage ($U$) is not definitive, and active time ($A$) provides the wrong conclusion, suggesting that VNF$_2$ is a bottleneck with 63% of the active time. Case #2 shows a shared bottleneck (both VNFs have similar impact on performance), with `guiltiness` leaning towards VNF$_2$ (the generally low values, 8% and 6%, can be justified by the response time, which is not as high as in case #1). Similarly to the previous case, $U$ is not definitive, with the difference that now $A$ correctly indicates VNF$_2$ as the bottleneck. Case #3 shows a scenario with an underprovisioned CPU capacity for VNF$_2$. The analysis based on `guiltiness`, as also the ones based on $U$ and $A$, properly identifies the bottleneck (VNF$_2$). Note that, if we use solely network queue usage ($Qu$) in these two previous cases, we would not be able to reach any conclusion about bottleneck VNFs. In case #4, there is no bottleneck, and thus all metrics confirm this behavior. Finally, when considering the average of all possible 81 combinations of resource allocation (by varying `cap` and `rate`), the `guiltiness` metric could detect 94% of bottleneck VNFs. On the other hand, observing metrics individually, $U$ detected 62.7%, $A$ detected 61.2%, and $Qu$ detected only 58.2% of bottlenecks. This occurs because methods based on CPU metrics ($U$ and $A$) are unsuitable for an underprovisioned network; and network queuing does not always occur if CPU is limited.

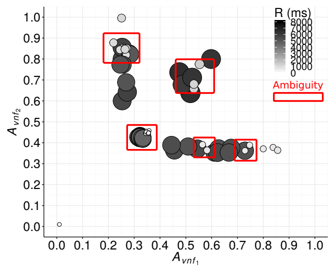
Table II
RESOURCE ALLOCATIONS AND MEASUREMENTS

| | Allocations | | | | Measurements | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | VNF$_1$ | | VNF$_2$ | | VNF$_1$ | | | | VNF$_2$ | | | | |
| # | CPU | Net | CPU | Net | $U$ | $A$ | $Qu$ | $G$ | $U$ | $A$ | $Qu$ | $G$ | $Rt$ |
| **1** | 10 | 10 | 10 | 50 | 0.23 | 0.51 | 0.4% | 0.5 | 0.22 | 0.63 | 0.0 | 0.07 | 6.5s |
| **2** | 10 | 50 | 10 | 50 | 0.25 | 0.54 | 0.0 | 0.06 | 0.25 | 0.72 | 0.0 | 0.08 | 1.4s |
| **3** | 100 | 100 | 10 | 100 | 0.02 | 0.25 | 0.0 | 0.03 | 0.28 | 0.99 | 0.0 | 0.11 | 1.0s |
| **4** | 100 | 100 | 100 | 100 | 0.03 | 0.35 | 0.0 | 0.04 | 0.03 | 0.44 | 0.0 | 0.05 | 0.24s |

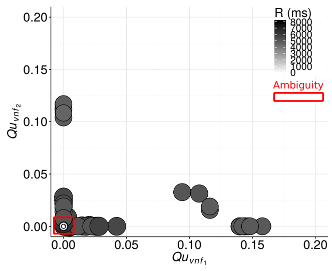
We also evaluate how the adaptive system works to adjust the coefficients and how it affects `guiltiness`. We normalize the measured response times and plot them against the estimated $G_c$ (Eq. 5) in Fig. 6. Fig. 6(a) shows the computed values using the default weights, whereas Fig. 6(b) depicts the values using the coefficients defined by the learning algorithm. The central line represents a perfect fit between the values



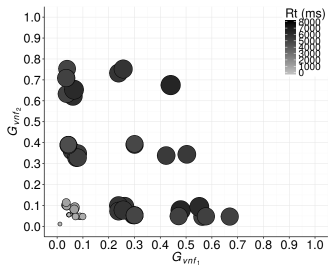
(a) CPU usage (as in [13])



(b) active time (proposed by [11])



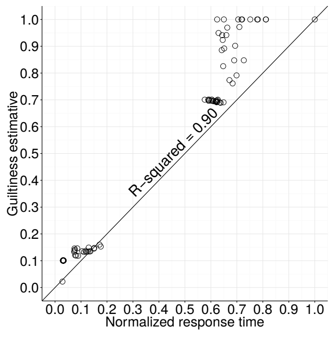
(c) vNIC queue (in line with [4])



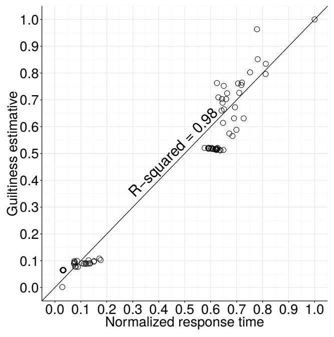
(d) `guiltiness`

Figure 5. Response time versus metric measurements in each VNF. Each point (x,y) corresponds to a given metric measurement in both VNFs and red boxes denote ambiguous situations

estimated by the model and those measured in the experiments. Results reveal that the default weights provide a pessimistic prediction. Using the default weights, `guiltiness` estimation starts to move up from the normalized response time at $RtNorm = 0.7$. As a consequence, the default coefficients provided an R-squared equal to $0.90$, whereas the adaptive system provides a better fit, with an R-squared equal to $0.98$.



(a) Default weights



(b) Learned weights

Figure 6. Model fit according to default and learned coefficients.

## B. Results reproduction

We provide artifacts for researchers to reproduce our experiments. These artifacts allow both reproducibility and repeatability of results. Results reproducibility denotes the ability to run the experiments and to obtain similar results. Results repeatability contains steps to plot the charts presented in this paper, by using the same data from our environment. We have tested these artifacts in the Linux distribution Ubuntu 14.04.5 LTS, with kernel 3.13.0-101-generic. To present a more straightforward reproduction process, we emulated an NFV network through the SONATA emulation platform [20] in a VM, in which VNFs run as docker containers. To perform these experiments, one must access the experiment page published at GitHub[3] and follow the provided instructions.

## C. Evaluation discussion

One can argue that the evaluated scenarios are limited to the size of chains. We defend that a chain with two forwarding VNFs is representative for general NFV scenarios, since the behaviors we observed will sustain with a larger number of VNFs. Also, scenarios are similar to the ones presented in related work. Both Wu *et al.* [4] and Gember *et al.* [12] evaluated their approaches in scenarios with the same size as ours. In the former, the bottlenecks were created through synthetic programs and solved through resource scaling actions. In the latter, network capacity limitations forced bottleneck in individual VNFs.

[3]https://github.com/ricardopfitscher/genic-experiment

## VII. Conclusions and Future Work

In NFV environments, special client demands can be met by creating customized service chains that can contain a variety of network functions. Bottlenecks on such service chains directly impact on the performance of end-services, and the literature exhibits several metrics to analyze the VNFs performance. In this paper, we argued that the results provided by such metrics are inconclusive to point out bottlenecks because of the heterogeneity of network functions. We also argued that bottlenecks can be detected by estimating the impact on the processing delay caused by each VNF in the service chain.

We proposed the `guiltiness` metric, which consists of a mathematical function that combines, in a weighted sum, metrics from queueing theory and empirical observations. To adjust the `guiltiness` values to the environment particularities, we proposed a monitoring architecture and a learning algorithm that monitors service chains, and adjust the model coefficients, respectively. Through experimental evaluations, we conclude that `guiltiness` outperforms the traditional metrics when characterizing the performance degradations caused by resource limitations: while traditional metrics struggled to represent the performance, `guiltiness` estimates normalized response time with an R-squared value of $0.98$. We also confirm that estimating processing delay impacts is successful for detecting bottlenecks. While `guiltiness` identified 94% of bottlenecks, traditional metrics only detect around 63%, in the best case.

As future work, we will investigate how the model behaves when VNFs are processing concurrent flows, and when multiple service chains are sharing VNFs. Also, we plan to compare `guiltiness` to other performance indicators.

## References

[1] 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, pp. 236–262, 2015.

[2] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

[3] ETSI Group Specification, "Network Function Virtualisation; An Introduction, Benefits, Enablers, Challenges and Call for Action," European Telecommunications Standards Institute, Tech. Rep., october 2012.

[4] W. Wu, K. He, and A. Akella, "Perfsight: Performance diagnosis for software dataplanes," in *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, 2015, pp. 409–421.

[5] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network Function Virtualization: Challenges and Opportunities for Innovations," *Communications Magazine, IEEE*, vol. 53, no. 2, pp. 90–97, 2015.

[6] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and Precise Triggers in Data Centers," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 129–143.

[7] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica, "Multi-Resource Fair Queueing For Packet Processing," *ACM SIGCOMM Computer Communication Review*, vol. 42, pp. 1–12, 2012.

[8] P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf, "Naas: Network-as-a-service in the cloud," in *Presented as part of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*. San Jose, CA: USENIX, 2012. [Online]. Available: https://www.usenix.org/conference/hot-ice12/naas-network-service-cloud

[9] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An Analytical Model for Multi-tier Internet Services and Its Applications," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, 2005, pp. 291–302.

[10] F. Benevenuto, C. Fernandes, M. Santos, V. Almeida, J. Almeida, G. J. Janakiraman, and J. R. Santos, "Performance Models for Virtualized Applications," in *International Symposium on Parallel and Distributed Processing and Applications*. Springer, 2006, pp. 427–439.

[11] Y. Wang, Q. Zhao, and D. Zheng, "Bottlenecks in production networks: An overview," *Journal of Systems Science and Systems Engineering*, vol. 14, pp. 347–363, 2005.

[12] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar, "Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud," Tech. Rep., 2013.

[13] C. Sauvanaud, K. Lazri, M. Kaaniche, and K. Kanoun, "Anomaly Detection and Root Cause Localization in Virtual Network Functions," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016, pp. 196–206.

[14] C. Roser, M. Nakano, and M. Tanaka, "A practical bottleneck detection method," in *Proceedings of the 33nd conference on Winter simulation*. IEEE Computer Society, 2001, pp. 949–953.

[15] P. J. Denning and J. P. Buzen, "The Operational Analysis of Queueing Network Models," *ACM Computing Surveys (CSUR)*, vol. 10, pp. 225–261, 1978.

[16] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[17] N. J. Gunther, *Guerrilla Capacity Planning: a Tactical Approach to Planning for Highly Scalable Applications and Services*. Springer Science & Business Media, 2007.

[18] R. J. Pfitscher, E. J. Scheid, R. L. dos Santos, R. R. Obelheiro, M. A. Pillon, A. E. Schaeffer-Filho, and L. Z. Granville, "DReAM-A Distributed Result-Aware Monitor for Network Functions Virtualization," in *IEEE Symposium on Computers and Communication (ISCC)*, 2016, pp. 663–668.

[19] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, "Performance Comparison of Middleware Architectures for Generating Dynamic Web Content," in *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. Springer-Verlag New York, Inc., 2003, pp. 242–261.

[20] M. Peuster, H. Karl, and S. Van Rossem, "Medicine: Rapid Prototyping of Production-Ready Network Services in Multi-PoP Environments," *arXiv preprint arXiv:1606.05995*, 2016.