Adaptive & Learning-aware Orchestration of Content Delivery Services

Steven Van Rossem*, Thomas Soenen *, Wouter Tavernier*,
Didier Colle*, Mario Pickavet* and Piet Demeester*
*Ghent University – imec, IDLab, Department of Information Technology.
Email: wouter.tavernier@ugent.be

***Abstract*—Many media services undergo a varying workload, showing periodic usage patterns or unexpected traffic surges. As cloud and NFV services are increasingly softwarized, they enable a fully dynamic deployment and scaling behaviour. At the same time, there is an increasing need for fast and efficient mechanisms to allocate sufficient resources with the same elasticity, only when they are needed. This requires adequate performance models of the involved services, as well as awareness of those models in the involved orchestration machinery. In this paper we present how a scalable content delivery service can be deployed in a resource- and time-efficient manner, using adaptive machine learning models for performance profiling. We include orchestration mechanisms which are able to act upon the profiled knowledge in a dynamic manner. Using an offline profiled performance model of the service, we are able to optimize the online service orchestration, requiring fewer scaling iterations.**

***Index Terms*—VNF, NFV, Machine Learning, Performance Profiling**

I. Introduction

Enabled by Network Function Virtualization (NFV), softwarized implementations of network functions create very flexible deployment options. In function of the online workload, more or less virtualized datacenter resources (such as CPU cores and bandwidth) can be elastically allocated to the service. When a service provider offers a certain functionality to its users, the performance is specified in the Service Level Agreement (SLA). Typically, Key Performance Indicators (KPIs) such as response time, jitter or loss are given in the SLA, along with their acceptable boundary values. The service provider can however optimize the amount of allocated resources as long as this does not cause any unacceptable service degradation outside the SLA limitations. To optimize this process, the relation between performance and allocated resources should be known in advance, so the service provider can orchestrate efficiently, needing less scaling iterations. If we can model this relation, we can make a more accurate prediction of how much resources are needed to fulfil the SLA.

In this paper, we deploy a content delivery service chain (described in Section II), where the number of file servers can be scaled, following a certain workload pattern. In an offline development environment (similar to production), we first execute a series of tests while varying several workload and resource parameters. Using this pro-active measurement, we analyse how trends can be modelled, which allows us to predict the number of needed file servers, given a certain workload and response time. This model is later on transferred to the production environment, where the orchestration platform can make an informed prediction of the needed file servers to deploy. If this model was not available, a sequence of scaling operations would take place, until the service eventually reaches the required performance. This difference is experimentally validated. In Section IV we describe the process of profiling the service in an offline development environment and creating a performance prediction model. Later on, in Section V, we describe how the profiled model can be integrated in a NFV Management and Network Orchestration (MANO) platform used for operational service deployments.

II. The Content Delivery Service Topology

The service we use is doing basic content delivery. A load balancer uses a round-robin strategy to distribute incoming file requests to a number of servers. Each server can only be connected to the load balancer over a bandwidth-limited link (1Gbps). This represents a topology where the servers are deployed on less powerful environments and such as smaller edge compute nodes. This setup can for example represent a series of multiple mirror file servers, located at different locations for extra reliability and decentralization. The setup is depicted in Fig. 1.

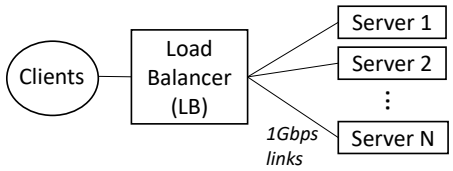


Fig. 1: The tested service topology scales between multiple file servers on limited network links.

We distinguish three different functional blocks:

a) Load Balancer: This is a Docker-based container. The implementation is Haproxy . We configure Haproxy as round-robin load balancer and enable the export of performance metrics such as response time.

b) File Server: This is a Docker-based container using the Python-based Flask implementation to serve files of random data and a given filesize.

c) Clients: The workload is generated in the Clients container. Locust is used as programmable file request generator. We generate a varying number of concurrent file requests of different filesizes.

III. Related Work

Our implementation is based on aspects of both VNF profiling and VNF orchestration. This section describes how our approach extends current related work. We base our modelling approach on the method used in [1]. We apply the method on a new service topology and investigate how we can extrapolate the modelled performance to untrained configurations. On the other hand, the authors in [2] and [3] propose to use analytical curve fitting to extrapolate service performance trends beyond trained workloads and resource allocations. The analytical trends are fitted online while the service is operating in the production environment. Our experiment however, tries to pro-actively model the performance trends, using an offline profiling environment. This helps to orchestrate the service more efficiently in production, and optimizes the learning time needed in production.

The MANO Framework orchestrates the end-to-end runtime life cycle of the profiled service in production. In order to use the profiled model in the operational context, the MANO Framework needs to be able to execute it. Therefore, the framework needs to support customisation, so that generic life cycle behaviour that is being applied to all services can be overwritten on a per service basis. In this context, [4] and [5] mention the necessity for MANO customisation but without proposing solutions. OSM [6] and Open Baton [7], two open source MANO Frameworks, allow developer-defined customisation for VNF life cycle events, such as attaching specific scripts to be executed when the VNF is started, configured or requires healing. However, both platforms miss the ability to tailor life cycle events on the network service level, e.g. custom procedures when a service should be scaled. ONAP [8] introduced a concept for a closed loop automation management platform (CLAMP) [9], allowing service level customisation through high-level policies. However, our profiled model is too complex to be captured by a hard coded, rule-based policy. SONATA-NFV [10] allows service developers to attach code snippets which customise the orchestration to the level of a modular service. Through Function (FSM) and Service Specific Managers (SSM), developers can package their code (including profiled models) as Docker containers and have them executed during the operational runtime of their services.

IV. Profiling Phase

Before we deploy the service in production, the topology (Fig. 1) is profiled in an offline environment. By profiling the service, we hope to derive a model to predict how many server instances should be allocated to the service, in order to meet a given workload and KPI target. We generate a series of workloads from the Clients to the LB, combined with a varying number of resource allocations. The monitored data is then later analysed to see if certain trends can be modelled.

As test hardware we use multiple equal compute nodes with 2x 8core Intel E5-2650v2 (2.6GHz) CPU running Ubuntu 18.04 as operating system. Linux Bridge is used as the hypervisor switch for the VNFs. We do not change default OS options (e.g. we leave hyperthreading enabled). The configuration options of Docker are called to isolate the CPU cores between the containers used in the tested topology.

A. Profiled Metrics

Monitoring data is the base for our modelling approach. During profiling, we monitor several metrics which we can group into three categories:

1) Workload parameters (wl):

Streams (s) : [1-1000] concurrent clients who are continuously generating file requests. So if there are n streams, there are always n concurrent file requests ongoing. 50 different values are selectively chosen, spaced evenly along the log scale.

Filesize : The size of the requested files is [0.5, 1, 5, 10] MByte.

2) Resource parameters (res):

Load Balancer cpu limit (LB_{cpu}) : The allocated vCPU share to the load balancer is [0.25, 0.5, 0.75, 1, 2, 3, 4, 5, 6] vCPU cores. This is only applicable in the offline profiling environment.

Number of servers (N_{servers}) : We create two datasets to analyse how well we can predict the performance of an extrapolated number of servers:

- N_{servers} = [2-5]: dataset for training the model.
- N_{servers} = [6-8]: dataset for testing the extrapolated model accuracy of an untrained number of servers.

3) Performance parameters (perf):

The service KPI is the **mean response time** (T) (ms) of the file requests, as measured at the ingress port of the Load Balancer. This is the average time it takes to complete a file download.

All combinations of the above parameters are tested, this yields a total of 12600 data points. This is then repeated at least 30 times, so we can average each datapoint over 30 measurements. This creates a good estimation of the average performance of this service topology. To speed up the measurements, we use the automated profiling environment described in [11]. This way we can run the tests in parallel, over multiple equal compute nodes. We show an example of the averaged profiled data in Fig. 2. The confidence intervals are drawn, but are barely visible because there is little variation in the measurements.

B. Resource Allocation Model

In order to predict how much resources should be allocated to the service, to meet a certain performance, we need to model the relation between workload (wl), performance KPI ($perf$) and resource allocation (res). Based on the work in [1], we can write following generic formulation to find the optimal resource allocation:

$$\begin{cases} f(wl, res) & \leq perf_{\text{target}} \\ wl & \geq wl_{\text{target}} \\ minimize & cost(res) \end{cases} \tag{1}$$

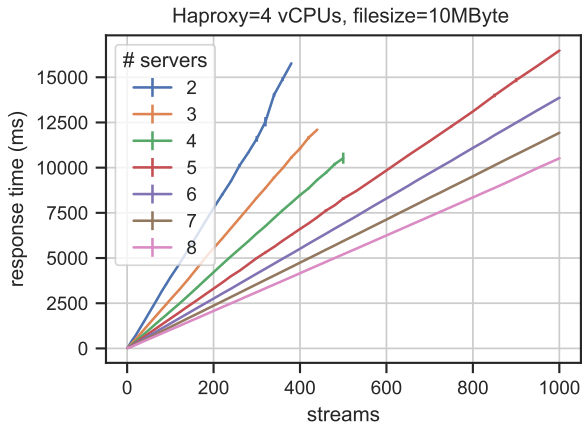


Fig. 2: Example of the profiled response time under a different number of servers.

The model $f(wl, res) = perf$ predicts the KPIs in $perf$ for a given workload and resource allocation. To find a solution for the optimization problem in Eq. 1 a simple heuristic can be used:
By iterating over all possible resource allocations, in order of increasing resource cost, we eventually find an allocation which satisfies the equations in Eq. 1. This means that the proposed resource allocation is the cheapest one to reach the target workload (wl_{target}) within the targeted performance limit ($perf_{\text{target}}$). When applied to our tested service topology, this yields the procedure given in Algorithm 1. The number of servers (N_{servers}) is increased until the targeted response time (T_{resp}) is reached, for a targeted number of streams (S_{target}). By using the model f we can in fact mimic a scaling sequence where more resources are added until the performance is within specification. In Table I we give an overview of the used variable names throughout the paper for easy reference.

S_{target}	Target number of streams
T_{target}	Target response time (upper limit)
N_{servers}	Number of servers
LB_{cpu}	Allocated vCPU of the load balancer
$filesize$	Filesize of the requests in the workload
f	Trained model to predict the response time T
T	A value of the mean response time
s	A value of workload $streams$
$s_{\max}$	Maximum number of streams, for a given target of N_{servers}, T_{target}, and $filesize$

TABLE I: Overview of used variable names.

Algorithm 1: Heuristic to find optimal N_{servers}

Data: S_{target}, T_{target}, $filesize$, f
Result: N_{servers}
$N_{\text{servers}} \leftarrow 2$ (initial N_{servers} allocation) ;
while $f(S_{\text{target}}, filesize, N_{\text{servers}}) \geq T_{\text{target}}$ **do**
  increase N_{servers} ;
end
return N_{servers};

The accuracy of the heuristic in Algo. 1 stands or falls with the accuracy of the model f. We investigate different methods for deriving f from our profiled dataset. The models are implemented using the Python-based library scikit-learn [12]:

- **ANN** The Artificial Neural Network (ANN) is a well known method from the machine learning domain. This method is capable of modelling very non-linear trends. When visually inspecting the trends for the response time (see Fig. 2), we expect however more linear relationships. As a compromise we use a less complex ANN, here with two hidden layers with 10 resp. 5 nodes.
- **Lasso** Since we expect more linear shaped trends, we also include the linear regression-based Lasso method. We also use polynomial expansion (second order) on the input parameters, in order to create some more degrees of freedom to fit the model.
- **Random Forest** The procedure of this method is to combine multiple decision trees in determining the final output rather than relying on an individually built decision tree. Maximum tree depth is set to 10, and the number trees in the forest is 100. The use of decision trees for modelling network service performance has been investigated in [13] with promising results. We include the method here for verification on our data sets.
- **Multivariate Linear Interpolation** Regression is done by interpolating linearly between surrounding samples. The interpolant is constructed by triangulating the input data using Delaunay triangulation, and on each triangle performing linear barycentric interpolation. This method also works in multiple dimensions, so the total workload and resource input space is taken into account to interpolate an intermediate KPI value. This method has been used for VNF modelling in [1]. We also include the method here for verification on our new datasets.
- **Multi Lasso** [1] This customized procedure is based on the method which showed most promising results in [1], used on the profiling of stand-alone Virtual Network Functions (VNFs). In [1], the method called 'Curve Fit' fits a number of analytic functions to the profiled data points. In our implementation, a Lasso model is trained (instead of an analytic function), separately for each profiled combination of ($filesize$ and N_{servers}), where only the number of $streams$ is varying as input parameter for the model. A prediction of a new workload value is made by using the previously mentioned Interpolation method between the Lasso models of surrounding nearby combinations (of $filesize$ and N_{servers}). One can think of this method as training a Lasso model for each of the lines in Fig. 2. Any intermediate value is then found by interpolation between the profiled Lasso models. More details of this method are given in [1].

In the following three subsections, we validate the usability of the above proposed methods in Eq. 1, hence providing an

accurate prediction of the needed number of servers in our service.

1) Extrapolate to untrained resource allocations: The usability of Eq. 1 and Algo 1 depends on the accuracy of the model $f(wl, res) = perf$ and its ability to extrapolate well to untrained values of N_{servers}. Only then can the model predict useful resource recommendations. We examine the different modelling methods to validate if the response time prediction can be extrapolated to untrained resource configurations. The accuracy of the methods is measured using the Root Mean Square Error (RMSE) of the predicted and the actual response time. The result is given in Fig. 3. In the left part, we can assess how each method can predict the response time for service configurations with $N_{\text{servers}} = [2, 3, 4, 5]$. We create five random splits with 90% of the dataset for training and 10% for testing. Then the RMSE is averaged over these five random data subsets. So in the left part, a lower RMSE indicates that the method can predict better the response time, as long as the test samples stay within the boundaries of the training data.

In the right part of Fig. 3, the RMSE is measured using the previous dataset (used on the left) as training data and the extrapolated dataset ($N_{\text{servers}} = [6, 7, 8]$) as test data. So the test data is not any more within the boundaries of the training data, as N_{servers} is increased. This indicates how well each method can predict the response time for N_{servers} beyond the training data.

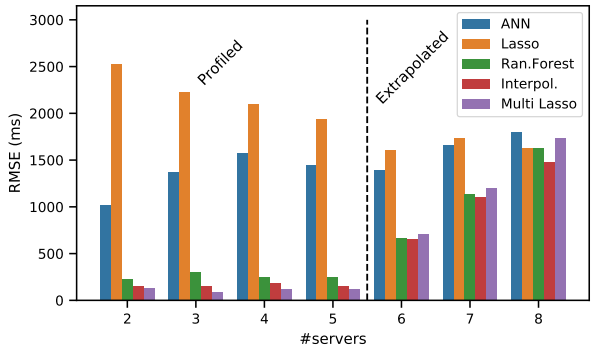


Fig. 3: Comparison of extrapolation capabilities between different modelling methods.

It is a known characteristic of many machine learning-based methods, that they do not predict well outside the boundaries of the training set. The accuracy of those methods is only good as long as there were enough training samples 'in the neighbourhood' of the test sample. In Fig. 3 we see that some methods (Random Forest, Interpolation and Multi Lasso) are better at interpolating data within the boundaries of the training dataset. When N_{servers} increases beyond the trained values, accuracy gets worse. Other methods, based on linear regression, can however extrapolate further (ANN, Lasso). We see in Fig. 3 that their accuracy does not worsen in the extrapolated area, but these methods are unfortunately considerably worse at the left side, interpolating values in within the boundaries of the training set. For ANN, we experience very large variations in the resulting accuracy, for repeated training cycles on the same training set. This indicates that accuracy depends on the random initialized weights in the model, and that there are too little training samples available for the model to converge. Decreasing the number of nodes in the ANN, would require less samples to train all the weights in the model, but at the cost a reduced trend fitting. We see in our test that smaller ANNs do not improve accuracy.

The Multi Lasso method has an unrivalled accuracy as long as it is tested within the boundaries of a training set. This is also aligned with the tests done in [1]. In the remainder of this paper we further improve the model to make predictions for an increasing number of servers.

2) Iterate through representative resource allocations: To find an optimal resource allocation in Eq. 1, our proposed heuristic iterates through possible resource allocations, ordered by increasing resource cost. To train the model f in a representative way, compared to the operational environment, we should align the samples taken in the offline profiling environment as much as possible. This means that we need to make sure that both workloads and allocated resources in the profiling tests are aligned with the ones used in the operational environment.

Concerning the allocated resources, an extra preprocessing step is taken to select representative measurements from the offline profiling tests. In our profiling environment, we have limited resources available. Therefore we carefully control how many vCPUs are allocated to the load balancer Haproxy (LB_{cpu}). For each profiled topology of N_{servers}, we increase LB_{cpu} until we are sure that the service is no longer constrained by LB. We define a resource cost as in Eq. 2 , where we take $w_1 = 0.1$ and $w_2 = 1$. This resembles that LB_{cpu} is a much cheaper resource, as it easier to allocate more vCPUs to LB, compared to adding an extra server.

$$cost(res) = w_1 * LB_{\text{cpu}} + w_2 * N_{\text{servers}} \tag{2}$$

The defined resource cost is a helpful tool to select from what LB_{cpu} onwards, the performance is limited by N_{servers}. This is exemplified in Fig. 4, where we plot the maximum streams possible, for a target $filesize = 10MB$ and response time $T_{\text{target}} = 5000ms$, in each profiled resource combination, ordered by increasing resource cost. To predict the maximum streams we use a Multi Lasso method, trained with samples of $N_{\text{servers}} \leq 5$. This results in a model f which can predict the response time T like in: $f(streams, filesize, N_{\text{servers}}, LB_{\text{cpu}}) = T$.

We look for the value of $streams$ at which T_{target} is reached, in each profiled resource combination of N_{servers} and LB_{cpu}. This is done by incrementing $streams$ in the above model until f reaches T_{target}. The *predicted trend* in Fig. 4 shows how this model f fits to the real measurements.

The crossed marker values or *profiled points* are found by interpolating the real measurements.

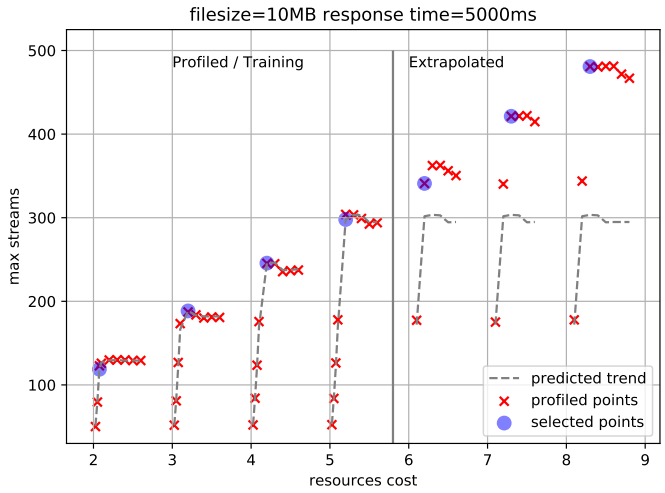


Fig. 4: Comparison of extrapolation capabilities between different modelling methods.

We use Fig 4 to illustrate following things:

- As expected, the Multi Lasso model fits well in the left part of Fig. 4, where nearby training samples are available. In the right part, f is stuck at the nearest sample included in the training set (this is a similar extrapolation behaviour as for the Random Forest or Interpolation methods).
- The resource cost increases slowly when adding more LB_{cpu}. We see in Fig. 4 that for each number of servers (the label on the x axis), the performance stabilizes and adding more LB resources does not help any more. This shows that from a certain LB_{cpu}, the response time is indeed limited by N_{servers} only. This effect is visible due to the selection of the weights in Eq. 2.

In the production environment we assume that LB is never the bottleneck. There, LB has no resource limitation and can freely use all of the available vCPUs. To predict this performance, we filter from the profiled data only the measurements which reflect an unsaturated LB: in Fig. 4 these are the large blue dots. These points are the selected resource allocations, where adding more LB_{cpu} does not improve the performance by more then 10%, for a given N_{servers}. As a result, we filter a set of profiled points, with a representative performance for an unconstrained LB and a specific $filesize$, T_{target} and N_{servers}.

3) Extrapolated Resource Recommendation :

It is clear that the procedure in Algo. 1 is not working to predict an extrapolated number of fileservers. This is due to the fact that the model f cannot extrapolate well to untrained settings. An improved version is proposed in Algo 2. This algorithm performs better because we do not use f directly to predict extrapolated values. We only use f within the boundaries of the training set: We illustrated in Fig. 4, that it is possible to derive a maximum number of streams for each *profiled* number of servers. So for a given T_{target} and $filesize$, we can use our profiled model f to derive an estimation for $s_{\max}$, the maximum number of streams reachable in every profiled number of N_{servers}. This way, we can gather several values for $s_{\max}$, which creates a new dataset, which is easier to extrapolate. For example, the *selected points* in Fig. 4 seem to be on a linear trend line. This reduced dataset is gathered in Line 1-6 of Algo. 2.

Line 4 in Algo. 2 uses an iterative solving algorithm to find a solution for $s_{\max}$. In our tests we use a brute force method which iteratively increments s by one until T_{target} is reached. Further optimized solver methods are possible but not tested here.

Algorithm 2: Heuristic to find optimal N_{servers} (with extrapolation beyond profiled data)

Data: S_{target}, T_{target}, $filesize$, f
Result: N_{servers}

1 $N_{\text{profiled}} \leftarrow$ profiled values of N_{servers} ;
2 $P \leftarrow$ empty dataset ;
3 **foreach** $N_i \in N_{\text{profiled}}$ **do**
4 $\quad s_{\max} \leftarrow$ solve $f(s, filesize, N_i) = T_{\text{target}}$;
5 $\quad P.append([s_{\max}, N_i])$;
6 **end**
7 $f_{regr} \leftarrow LinearRegression(s, N_i \in P)$;
8 $N_{\text{servers}} \leftarrow ceil(f_{regr}(S_{\text{target}}))$
9 return N_{servers};

Figure 5 demonstrates how a trend for the required number of servers was derived using Algo 2. The blue *profiled points* are derived from the training dataset of $N_{\text{servers}} = [2-5]$. These points are used in Line 7 in Algo. 2 to train a linear regression. In our experiment we assume a linear trend is existing between $s_{\max}$ and N_{servers}. We achieve a stepwise prediction by ceiling the regression line to the upper integer value. The orange *new points* indicate points outside of the training set, these are measured in the topology with $N_{\text{servers}} = [6-8]$. We see that these points are well on the predicted linear trend.

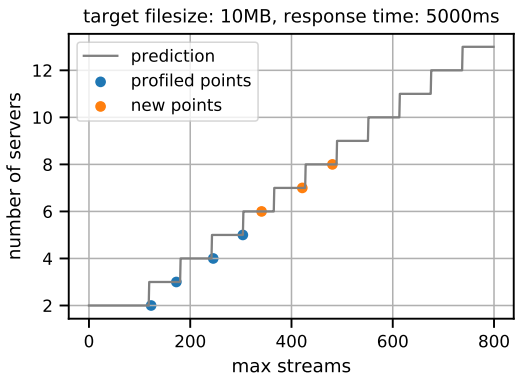


Fig. 5: Linear extrapolation of the profiled dataset.

C. Other Modelling Approaches

In this subsection we why other profiling methods would yield less accurate results. A general machine learning-based approach would be to consider purely the profiled values of $filesize$, $streams$ and T to predict N_{servers}, as in Eq. 3.

$$F(filesize, streams, T) = N_{\text{servers}} \tag{3}$$

The above modelling approach has several disadvantages:

- The parameter N_{servers} is not continuous but categorical in nature. Several values of $filesize$, $streams$ and T map to a single integer value of N_{servers}, as seen in Fig. 5. For a generic machine learning method, this would mean that the training samples are following a step-like shape. We know that regression-based methods are not good at modelling such discontinuities, certainly when the number of training samples is rather limited, as in our use case.
- When considering classifier-based methods, we could successfully train a model to predict N_{servers} classes, but the number of classes is limited to the values of N_{servers} in the training set. Trained classifiers can therefore not extrapolate and are bounded to predict only trained values of N_{servers} (similar to the faulty predicted values in the extrapolated part of Fig. 4).

The above reasons further explain why generic machine learning methods cannot be used *directly*. The approach in Algo. 2 effectively bypasses the explained shortcomings of machine learning methods in our profiling use case. We can further note that Algo. 2 can be easily adapted to model other trend lines if needed. Line 7 in Algo. 2 can be easily replaced by another regression method, if the trend line seems to be of a different, non-linear shape. For modelling the response time, a linear trend is also what is intuitively expected : From a certain number of concurrent streams, the service's resources are saturated and the maximum bandwidth ($BW_{\max}$) is reached. This maximum bandwidth is shared among the increasing number of streams, each downloading the same filesize. This leads to a response time T as given in Eq. 4 :

$$T = \frac{filesize}{BW_{\max}} \; streams \tag{4}$$

We can additionally assume that $BW_{\max}$ is strongly correlated with N_{servers} in our service. If Eq. 4 is true, this would mean that the model accuracy must improve if we use $1/N_{\text{servers}}$ as a feature. The product $filesize * streams * 1/N_{\text{servers}}$ should then be a good predictor for T. We test this assumption by testing including $1/N_{\text{servers}}$ in the training set of the Lasso method, and use polynomial expansion of the third order to ensure that the input feature set is expanded with also the product of all three features as mentioned. We show the prediction accuracy of this $Lasso3$ method in Fig. 6. We see that the $Lasso3$ method is indeed outperforming the earlier used Lasso method, which indicates that the relation of Eq. 4 improves the prediction model. But bigger problems occur when using $Lasso3$ for extrapolation, probably because the third order polynomial expansion is overfitting the training set,

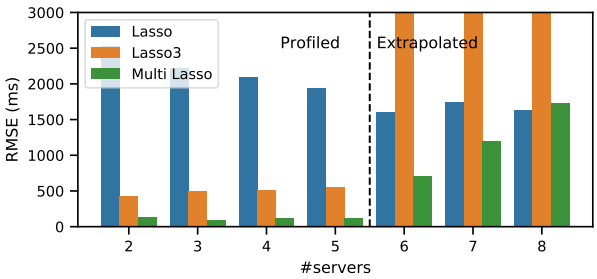


Fig. 6: Accuracy of the Lasso3 method.

creating non-linear trends. We can conclude from the results in Fig. 6 that linear regression-based methods can be improved greatly, if expert knowledge as in Eq. 3 is used. However our proposed Multi Lasso method remains the best choice for our use case.

V. MANO Platform Integration

To perform automated life cycle management for content delivery services at runtime, one uses NFV Management and Orchestration (MANO) frameworks. These frameworks typically automate life cycle decicions such as instantiation, scaling, migration, configuration and termination, based on a high level service description. For example, the service description of a content delivery service could contain a scaling policy indicating that the service should scale out when the average response time of the service exceeds $0.05s$, and scale in if it falls below $0.03s$. It is then the task of the MANO Framework to enforce this policy by monitoring the associated metrics and automatically trigger scaling events when the thresholds are crossed.

In this paper we focus on the SONATA-NFV [10] MANO Framework, as argumented in Section III, since it contains a unique feature that is required for our setup. It allows the

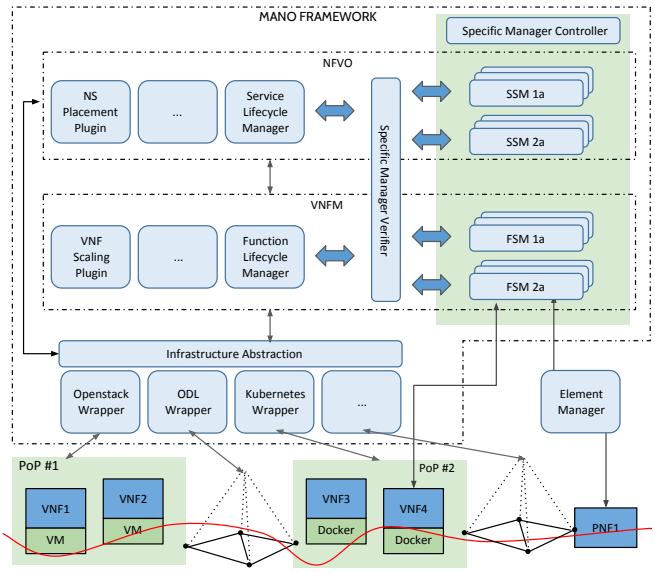
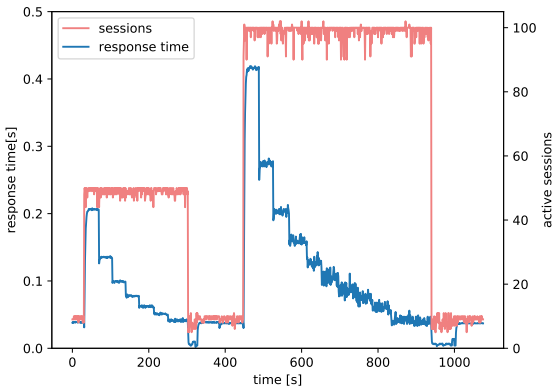


Fig. 7: SONATA-NFV MANO framework Architecture with Specific Managers.

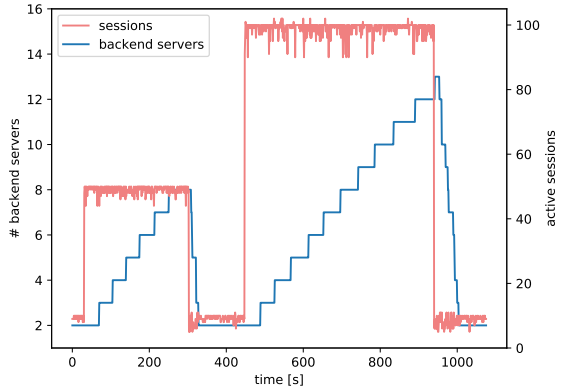
developer of the content delivery service to add actual code to its service description, which can be executed by the MANO Framework at the appropriate time. With this feature, service developers get a more finegrained control over the life cycle management of their services at runtime. Where earlier, they were limited to describe the desired behaviour through high level policies, they can now include (machine learning-based) prediction models to alert the MANO Framework when life cycle events are needed.

The SONATA-NFV MANO Framework enables this feature through the use of SSMs. These are Docker containers, provided by a service developer, that are deployed and attached as part of the control plane when the content delivery service is being deployed. During default life cycle processes (e.g. instantiation and termination), the MANO Framework will interact with the SSM to check if there is any service specific control behaviour that needs to be considered. The MANO Framework also allows the SSM access to all monitoring data related to the associated content delivery service, and provides a hook so that the SSM can send instructions (e.g. to scale or to migrate) to the MANO Framework. Therefore, at runtime, the SSM can process all the relevant monitoring data, use it as input for the machine learning model that was copied into the SSM container, and request a scaling event from the MANO Framework whenever the model indicates one is needed. Figure 7 shows how the SSM concept relates to the entire architecture of the SONATA-NFV MANO Framework. For more details on this architecture, see [10].

We have ran two experiments to show the added value of including the profiling model as code to the service description. Each scenario applies a different scaling logic to a content delivery service containing a Haproxy load balancer and a set of Python Flask backend fileservers, as described earlier. The number of fileservers is to be decided by the scaling logic. A SONATA-NFV MANO Framework instance is used to orchestrate this service on top of a Kubernetes cluster, with 32 Intel(R) Xeon(R) CPU E5-2650 v2 CPUs and 32 GB of RAM available. Load is generated using the Python Locust tool by requesting 1 MB files in concurrent threads through the REST API that is exposed by the service. In the first scenario, we apply a simple high level scaling policy that instructs the MANO Framework to add a single fileserver if the average response time of the service exceeds $0.05s$, and to remove a single fileserver it falls below $0.03s$. In the second scenario, we deploy the service together with an SSM that contains our pre profiled model f. By using Algo 2, the SSM can plugin monitored data of the number of concurrent sessions (S_{target}), the filesize being requested and the cut-off response

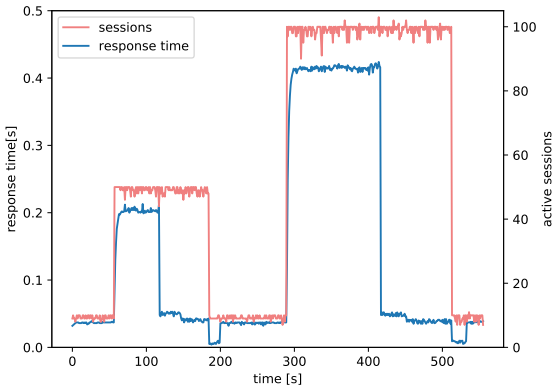


(a) response time

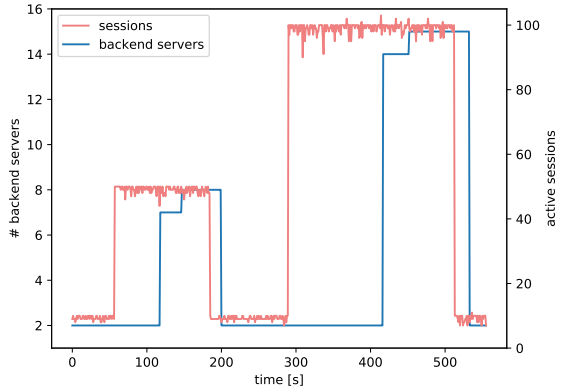


(b) Number of backends

Fig. 8: Scenario 1: Runtime scaling without learning



(a) response time



(b) Number of backends

Fig. 9: Scenario 2: Runtime scaling with learning (Note the different x axis scale compared to Fig. 8)

time (T_{target}), and calculate when the service should be scaled.

The results of both scenarios are shown in Figures 8 and 9. They show for each scenario how the service response time and the number of backend servers evolve when the scaling logic is being applied by the MANO Framework. The shown data is collected from the Haproxy load balancer. The scaling regime was triggered by varying the number of concurrent requests generated by the Locust tool, which is recorded by the load balancer as number of active sessions. When the number of active sessions increases, we see that for scenario 1 the number of backend fileservers gradually increases, resulting in a gradual decrease of the response time. In scenario 2, the desired number of backend fileservers is reached much quicker, because the profiling model provided a good estimation of how many fileservers are needed for the new conditions. This results in a shorter scaling procedure so that the required QoS in terms of service response time ($< 0.05s$) is reached much faster (only 2 scaling iterations versus 11 for a surge of 10 active sessions to 100).

VI. Conclusion and Future Work

Profiling a network service implies that a set of varying workload parameters and resource allocations is tested in an isolated environment, prior to deployment in production. This pro-active, offline test allows to monitor and model the service's performance. The creation of such a performance model has been exemplified on a content delivery service, where a load balancer distributes incoming file requests to a set of bandwidth- limited servers. The goal of the profiled model is to predict how much servers need to be reserved in the production environment, to meet a given SLA specification of max concurrent streams and response time. We also showed how a provided performance model can be integrated in production-grade MANO platform architectures. We see however following paths for improvement:

A. Online Learning as MANO Platform Extension

As shown in Figure 9, the scaling data provided by the profiling model is not perfect. Both for the increase from 10 to 50 sessions, and from 10 to 100, Figure 9b shows that after one initial large scale out event of multiple backends, estimated by the model, the MANO decides to scale out one more time. This secondary scale out event is triggered by the fact that the specified response time threshold of $0.05s$ was not yet reached. The scaling information from the model was good, but still deviated a bit from the actual scenario. This can be explained by looking at the used environments. It is very hard for offline profilers to mimic the exact operational environment when they are collecting profiling data in their setup. Therefore, follow-up procedures should always be in place when earlier profiled data is being used at runtime. One such procedure is the application of online learning. More specifically in our workflow, this comes down to retraining model f with newly gathered data and plugging it back into Algo 2. In the MANO framework, the SSM contains the profiled model and can receive new monitoring data associated to the content delivery service. Next, this data could be used to retrain the model and further improve the scaling behaviour in the specific operational environment currently being used.

B. Hybrid Modelling Approaches

One of our learnings is that standard machine learning methods fall short to extrapolate service performance to untrained resource allocations. Either they are too complex to be trained by the limited amount of profiled samples, or they only work well within the boundaries of the training data. To overcome this, we needed to filter out our wanted performance metrics from profiled resource allocations only (Algo. 2). On this simplified dataset we can do a linear regression, in order to extrapolate performance beyond profiled settings. In our example we witnessed a clear linear trend to extrapolate the performance. More tests on varying types of network services are needed to check if other (non-) linear KPI trends are occurring. Also other types of KPIs such as loss or jitter could show different trend shapes. Then it is the question if and how existing machine learning methods can be used for online retraining of non-linear service performance trends. Our presented example use-case and followed methodology can trigger further research in this area.

Acknowledgement

This work has been performed in the framework of the NGPaaS and 5GTANGO project, funded by the European Commission under the Horizon 2020 and 5G-PPP Phase2 programmes, resp. under Grant Agreement No. 761 557 and 761 493 (http://ngpaas.eu) (https://www.5gtango.eu). This work is partly funded by UGent BOF/GOA project 'Autonomic Networked Multimedia Systems'.

References

[1] S. Van Rossem, W. Tavernier, D. Colle, M. Pickavet, and P. Demeester, "Profile-based resource allocation for virtualized network functions," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2019.

[2] J. O. Iglesias *et al.*, "Orca: an orchestration automata for configuring vnfs," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, 2017, pp. 81–94.

[3] P. Xiong, C. Pu *et al.*, "vperfguard: an automated model-driven framework for application performance diagnosis in consolidated cloud environments," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ACM, 2013, pp. 271–282.

[4] S. Van Rossem *et al.*, "Introducing development features for virtualized network services," *IEEE Communications Magazine*, 2018.

[5] R. Szabo *et al.*, "Elastic network functions: opportunities and challenges," *IEEE network*, vol. 29, no. 3, pp. 15–21, 2015.

[6] "OSM Release Six Documentation," https://osm.etsi.org/wikipub/index.php/OSM_Release_SIX_Documentation, (Accessed Feb. 18, 2020).

[7] G. Carella *et al.*, "Prototyping nfv-based multi-access edge computing in 5g ready networks with open baton," in *IEEE NetSoft*, 2017, pp. 1–4.

[8] "ONAP: Open Network Automation Platform," https://www.onap.org/architecture, (Accessed Feb. 18, 2020).

[9] ONAP, "Documentation of ONAP's CLAMP module," http://onap.readthedocs.io/en/latest/submodules/clamp.git/docs/index.html, (Accessed Oct. 10, 2018).

[10] T. Soenen *et al.*, "Empowering network service developers: enhanced nfv devops and programmable mano," *IEEE Communications Magazine*, vol. 57, no. 5, pp. 89–95, 2019.

[11] S. Van Rossem, W. Tavernier, D. Colle, M. Pickavet, and P. Demeester, "Optimized Sampling Strategies to Model the Performance of Virtualized Network Functions," Jan. 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02354401

[12] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[13] M. Peuster and H. Karl, "Understand your chains and keep your deadlines: Introducing time-constrained profiling for nfv," in *2018 CNSM Conference*, Nov 2018, pp. 240–246.